Teoretyczne Podstawy Informatyki

Mgr inż. Janusz Marecki

WSTĘP

.....

.....

.....

Spis treści

1. WPROWADZENIE 8

1.1 DOWODY INDUKCYJNE 8
1.2 DOWODY WTEDY-I-TYLKO-WTEDY 9
1.3 JĘZYKI 11
1.4 AUTOMAT SKOŃCZONY 12

2. AUTOMATY SKOŃCZONE 13

2.1 FORMALNA DEFINICJA AUTOMATU SKOŃCZONEGO 13
2.2 ROZSZERZENIE TRANZYTYWNE FUNKCJI δ 14
2.3 NIEDETERMINISTYCZNY AUTOMAT SKOŃCZONY 15
2.4 AUTOMAT SKOŃCZONY I ε-ELIMINACJA 18
2.5 AUTOMAT MINIMALNY 19

3. WYRAŻENIA REGULARNE 22

3.1 OPERATORY I OPERANDY 22
3.2 RÓWNOWAŻNOŚĆ AUTOMATÓW SKOŃCZONYCH I WYRAŻEŃ REGULARNYCH 23
3.3 PRAWA ALGEBRAICZNE DLA WYRAŻEŃ REGULARNYCH 26
3.4 SPRAWDZANIE PRAW DLA WYRAŻEŃ REGULARNYCH 27
3.5 LEMAT O POMPOWANIU 27
3.6 WŁASNOŚCI RODZINY JĘZYKÓW REGULARNYCH 29
3.7 PROBLEMY DECYZYJNE DLA JĘZYKÓW REGULARNYCH 31

4. PODSTAWY JĘZYKÓW BEZ-KONTEKSTOWYCH 34

4.1 GRAMATYKI 34
4.2 AUTOMAT ZE STOSEM 39
4.3 RÓWNOWAŻNOŚĆ GRAMATYK BEZ-KONTEKSTOWYCH I AUTOMATÓW ZE STOSEM 43
4.4 DETERMINISTYCZNY AUTOMAT ZE STOSEM 46

5. ANALIZA JĘZYKÓW BEZ-KONTEKSTOWYCH 48

5.1 CZYSZCZENIE GRAMATYK BEZ-KONTEKSTOWYCH 48
5.2 LEMAT O POMPOWANIU DLA GRAMATYK BEZ-KONTEKSTOWYCH 52
5.3 WŁASNOŚCI RODZINY JĘZYKÓW BEZ-KONTEKSTOWYCH 55

6. JĘZYKI KONTEKSTOWE I MASZYNA TURINGA 61

6.1 PODZIAŁ GRAMATYK CHOMSKY'EGO 61
6.2 DEFINICJA MASZYNY TURINGA (MT) 62
6.3 RODZAJE MASZYN TURINGA 66

7. FUNKCJE REKURENCYJNE 72

7.1 TEZA CHURCHA 72
7.2 RODZAJE FUNKCJI REKURENCYJNYCH 72
7.3 FUNKCJE REKURENCYJNIE PRYMITYWNE 74
7.4 KONSTRUKCJA FUNKCJI REKURENCYJNIE PRYMITYWNYCH 76
7.5 KODOWANIE CIĄGÓW 79
7.6 REKURENCJA NIE PRYMITYWNA 82

8. OBLICZALNOŚĆ I ZŁOŻONOŚĆ OBLICZENIOWA 83
8.1 OBLICZALNOŚĆ A MASZYNA TURINGA 83
8.2 MASZYNA TURINGA JAKO MODEL OBLICZEŃ 84
8.3 RACHUNEK λ 87
8.4 KLASY ZŁOŻONOŚCI 93
8.5 NP-ZUPEŁNOŚĆ 95
9. KLASY ALGORYTMÓW 98
9.1 BRANCH & BOUND 98
9.2 PROGRAMOWANIE DYNAMICZNE 103
9.3 DZIEL I ZWYCIĘŻAJ 106
9.4 ALGORYTMY ZACHŁANNE 108
9.5 ALGORYTMY Z NAWROTAMI 111
10. PYTANIA KONTROLNE 114
11. ZAKOŃCZENIE 118

Spis rysunków

Rysunek 1 Automat skończony akceptujący liczby rzeczywiste 12
Rysunek 2 Automat akceptujący słowa o parzystej liczbie 0 i 1. 14
Rysunek 3 Przykładowy automat niedeterministyczny 16
Rysunek 4 Automat przed ε-eliminacją 18
Rysunek 5 Automat po ε-eliminacji 19
Rysunek 6 Automat przed minimalizacją 20
Rysunek 7 Automat zminimalizowany 21
*Rysunek 8 Automat dla wyrażenia regularnego (0(0+1))** 24
Rysunek 9 RE generowane na podstawie automatu 26
Rysunek 10 Ilustracja lematu o pompowaniu 28
Rysunek 11 Drzewo wywodu dla gramatyki bez-kontekstowej 37
Rysunek 12 Automat ze stosem 40
Rysunek 13 Eliminacja produkcji zbędnych 52
Rysunek 14 Lemat o pompowaniu dla gramatyk bez-kontekstowych 54
Rysunek 15 Pokrycie płaszczyzny kolejnymi liczbami naturalnymi 80
Rysunek 16 Kodowanie maszyny Turinga 85
Rysunek 17 Proces arytmetyzacji 87
Rysunek 18 Drzewa reprezentujące termy 91
Rysunek 19 Zmienne wolne i połączone 92
Rysunek 20 Zmiana nazw zmiennych w drzewie 92
Rysunek 21 Klasyfikacja problemów o różnej złożoności obliczeniowej 96
Rysunek 22 Graf połączeń dla komiwojażera i najkrótsza marszruta 99
Rysunek 23 Stany wież Hanoi: początkowy i końcowy 107
Rysunek 24 Zachłanne generowanie marszruty komiwojażera 109
Rysunek 25 Labirynt i metoda z nawrotami 111

1. Wprowadzenie

Streszczenie: *Większość dowodów twierdzeń w informatyce korzysta z metody indukcji matematycznej. W rozdziale tym przedstawimy formalną definicję indukcji, a następnie przeprowadzimy przykładowy dowód indukcyjny. W teorii języków i automatów, której dotyczą późniejsze rozdziały istotne znaczenie ma pojęcie równoważności zbiorów, stąd w dalszej części tego rozdziału pokażemy jak udowadnia się równoważność. Na końcu rozdziału zademonstrujemy przykładowy język oraz prosty aparat skończony.*

1.1 Dowody indukcyjne

Załóżmy, że dana jest rodzina obiektów X (liczby całkowite, drzewa, itp.). Ponadto, niech S(x) będzie własnością, którą może posiadać obiekt $x \in X$. W celu udowodnienia S(y) $\forall y \in X$ należy:

1. Określić bazę indukcji: udowodnić *wprost* S(y) dla jednej lub kilku małych wartości y.
2. Wykonać krok indukcyjny: zakładając, że spełnione jest S(y) dla y „mniejszego od x" udowodnić S(x).

Przykład:

Udowodnij, że każde drzewo binarne posiadające n liści składa się z 2n-1 węzłów:
Dowód: □

- Dla każdego drzewa T, własność S(T) zdefiniujemy następująco:
  S(T): jeśli T jest drzewem binarnym o n liściach to T składa się z 2n-1 węzłów.
- Indukcja oparta jest na ilości węzłów drzewa T.
- Dowód indukcyjny:
  - *Baza*: dla drzewa składającego się z 1 węzła, węzeł ten jest jedynym liściem: 1 = 2*2 –1.
  - *Krok indukcyjny*: załóżmy, że S(T) jest spełnione dla drzew T posiadających *co najwyżej* n węzłów. Przeanalizujmy drzewo T składające się z n+1 węzłów. Zgodnie z własnością drzew binarnych, T jest drzewem składającym się z dwóch poddrzew binarnych T_l, T_p, przy czym zarówno T_l jak i T_p, posiadają co najwyżej n węzłów. Przyjmując r za korzeń drzewa T , mamy więc

następującą sytuację: $T_l \leftarrow r \rightarrow T_p$. Jeśli u = ilość liści drzewa T_l oraz v = ilość liści drzewa T_p, wówczas w = ilość liści drzewa T = u + v. Z założenia indukcyjnego T_l oraz T_p posiadają odpowiednio 2u-1 oraz 2v-1 węzłów, zatem T posiada 1 + (2u-1) + (2v-1) węzłów = 2*(u+v) –1 = 2w – 1 węzłów. ■

1.2 Dowody Wtedy-i-Tylko-Wtedy

Często spotykamy się z sytuacją, gdy należy udowodnić zdanie „X wtedy i tylko wtedy gdy Y". W tej sytuacji należy osobno udowodnić dwa zdania logiczne:

- Zakładając X dowieść Y
- Zakładając Y dowieść X

Przy dowodzeniu zdań Wtedy-i-Tylko-Wtedy (równoważności) nie należy zapominać o tym, że:

- Zakładając X, Y możemy dowieść tylko w przypadku gdy X jest zdaniem prawdziwym.
- Zdaniem równoważnym zdaniu „Jeśli X to Y" jest zdanie „Jeśli nie Y to X". Drugie zdanie jest kontrapozycją zdania pierwszego.

Równoważność zbiorów:

Wiele twierdzeń w teorii języków i automatów dotyczy zbadania równoważności zbiorów zapisanych na dwa różne sposoby. Przykładowo, jeśli chcemy zbadać równość zbiorów X i Y, należy udowodnić zdanie logiczne:

- „$\forall z$: $z \in X$ wtedy i tylko wtedy, gdy $z \in Y$", co sprowadza się do dwóch dowodów:
  - Załóżmy że $z \in X$, dowieść że $z \in Y$, oraz
  - Załóżmy że $z \in Y$, dowieść że $z \in X$.

Przykład: równowaga nawiasów

Poniżej przedstawione są dwie definicje „równowagi nawiasów"

1. *Gramatyczna*:
   a. Pusty ciąg znaków ε jest zrównoważony.
   b. Jeśli ciąg w jest zrównoważony, to ciąg (w) jest także zrównoważony.

c. Jeśli w_1 oraz w_2 są zrównoważone, to zrównoważony jest także w_1w_2.

2. poprzez *Skanowanie*:
   a. Ciąg w posiada taką samą liczbę prawych i lewych nawiasów.
   b. Każdy prefiks ciągu w posiada co najmniej tyle nawiasów lewych co prawych.

Równowaga gramatyczna (RG) oraz „poprzez skanowanie" (RS) mogą być własnością dowolnego ciągu znaków. Następujące twierdzenie jest prawdziwe:

Ciąg w posiadający nawiasy jest RG wtedy i tylko wtedy, gdy jest RS.

Dowód:□

Jeśli RS to RG (RG ⇐ RS)

Załóżmy że w jest zrównoważony poprzez skanowanie; Udowodnimy, że jest zrównoważony gramatycznie. Dowód indukcyjny ze względu na |w| (długość ciągu w).

- *Baza*: Jeśli |w| = 0, czyli w = ε , wówczas w jest RG - warunek a.
  Warto zauważyć, że nie rozpatrujemy tutaj nawet czy w jest RS (co jest prawdą).
- *Krok indukcyjny*: Załóżmy, że RS ⇒ RG jest prawdziwe dla ciągów krótszych od w. Weźmy dowolny niepusty ciąg w spełniający RS
  1. Przypadek: w nie ma niepustego prefiksu posiadającego taką samą liczbę „(" i „)". Stąd wniosek, że w musi się zaczynać od „(" gdyż inaczej prefiks długości 1 posiadałby taką samą liczbę „(" i „)". Ponadto w musi kończyć się na „)" gdyż inaczej prefiks długości |w|-1 posiadałby taką samą liczbę „(" i „)". Z tego wniosek, że w ma postać w = (x). Teraz z kolei x ma własność RS, gdyż x posiada taką samą liczbę nawiasów prawych i lewych oraz każdy prefiks x posiada co najwyżej tyle „(" co „)". Z kolei x jest ciągiem krótszym od w stąd (założenie indukcyjne) x ma własność RG. Teraz jeśli x ma własność RG, to z warunku b RG mamy (x) = w ma własność RG.
  2. Przypadek: w = xy, gdzie x jest najkrótszym niepustym prefiksem w posiadającym taką samą liczbę „(" co „)", oraz y≠ε. Łatwo sprawdzić, że zarówno x jak i y posiadają własność RS. Ponadto, x i y są krótsze od w więc z założenia indukcyjnego x oraz y mają własność RG. Stąd w=xy ma własność RG (warunek c. definicji równoważności gramatycznej). ■

Dowód:□

Jeśli RG to RS (RG ⇒ RS)

Załóżmy że w jest zrównoważony gramatycznie; Udowodnimy, że jest zrównoważony przez skanowanie. Dowód indukcyjny ze względu na |w| (długość ciągu w).

- *Baza*: Jeśli |w| = 0, czyli w = ε , wówczas w spełnia warunki RG. Jak łatwo zauważyć ciąg pusty spełnia także 2 warunki RS.
- *Krok indukcyjny*: Załóżmy, że RG $\Rightarrow$ RS jest prawdziwe dla ciągów krótszych od w. Weźmy dowolny niepusty ciąg w spełniający RG:
  1. Przypadek: w ma własność RG poprzez warunek b. czyli w=(x). Ciąg x jest krótszy od w więc z założenia indukcyjnego x ma własność RS, z czego wynika:
     - x ma taką samą ilość „(” oraz „)”, czyli w=(x) ma taką samą ilość „(” oraz „)”.
     - Każdy niepusty prefiks x ma co najmniej tyle „(” co „)” czyli także każdy niepusty prefiks w=(x) ma co najmniej tyle „(” co „)”.

     Ostatecznie więc w=(x) posiada własność RS.
  2. Przypadek: w ma własność RG poprzez warunek c. czyli w=xy. Ciągi x oraz y są krótsze od w więc z założenia indukcyjnego oba mają własność RS. Teraz:
     - Musimy założyć, że ani x ani y nie są ε, bo wówczas jeden w nich byłby równy w, co z kolei nie dałoby nam możliwości rozpatrywania tego przypadku.
     - xy posiada taką samą liczbę „(” oraz „)” z założenia, że ten warunek spełnia zarówno x jak i y.
     - Jeśli w=xy posiadałby prefiks mający większą liczbę „)” niż „(”, wówczas:
       - jeśli ten prefiks byłby w całości prefiksem x, to x nie mógłby mieć własności RS – sprzeczność
       - jeśli ten prefiks zachodziłby na y, wówczas y nie mógłby mieć własności RS – sprzeczność

     Ostatecznie więc w=xy posiada własność RS.

     ■

1.3 Języki

- *Alfabet* jest to skończony zbiór symboli, np. {0,1} – alfabet binarny, kody ASCII
- *Słowo* jest skończonym ciągiem znaków z danego alfabetu, np. „1010” lub „babajaga”
- *Język* jest zbiorem słów zbudowanych na bazie pewnego alfabetu.

Warto zauważyć, że język może być zbiorem nieskończonym, jednak istnieje pewien skończony zbiór symboli, z których zbudowane są jego słowa.

Przykład języków:

- Zbiór wszystkich ciągów binarnych postaci $\{0^n1^n : n=1, ..., N\}$
- PASCAL – zbiór wszystkich dających się skompilować programów w PASCALU.
- Język polski

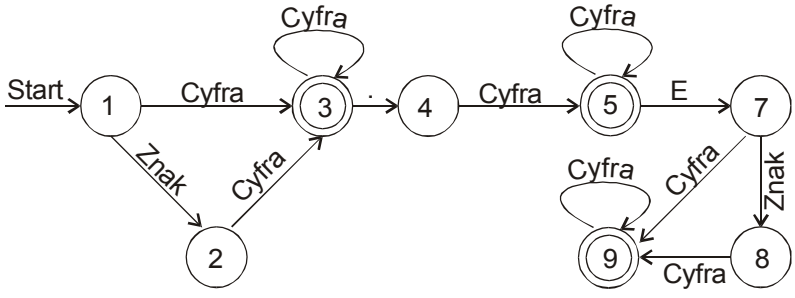
1.4 Automat skończony

Automat skończony jest ważnym sposobem zapisu pewnych prostych, jednak niezwykle użytecznych języków zwanych „*językami regularnymi*”. Na automat skończony składają się następujące elementy:

- Graf skierowany o skończonej liczbie wierzchołków zwanych *stanami*.
- Do każdego łuku przypisany jest jeden lub kilka symboli z alfabetu.
- Jeden stan jest wyodrębniony jako *stan startowy*.
- Niektóre stany są *stanami końcowymi* (*stanami akceptującymi*).
- Językiem generowanym przez automat skończony jest zbiór słów, które powstają przez wszystkie możliwe przejścia od stanu startowego do stanu końcowego zbierając po drodze symbole przypisane do łuków.

Przykład:

Poniższy automat akceptuje wszystkie liczby rzeczywiste zapisane w komputerze, np.: 31.89E-19. W związku z tym, że liczby mogą być całkowite / ułamkowe, ujemne / dodatnie wyróżniamy kilka stanów końcowych. Na rysunku nr 1 liczby w kółkach są oznaczeniem stanów. Stan reprezentowany przez dwa koncentryczne kółka jest stanem końcowym. Cyfra oznacza jeden z symboli ze zbioru {0,1,2,3,4,5,6,7,8,9}; Znak jest symbolem ze zbioru {-}.



Rysunek 1 Automat skończony akceptujący liczby rzeczywiste

2. Automaty skończone

Streszczenie: *W tym rozdziale szczegółowo omówimy automaty skończone. Rozpoczniemy od formalnej definicji automatu skończonego, pokażemy jak rozszerza się jego funkcję przejścia by można było mówić, że akceptuje on całe słowa, a nie tylko symbole oraz zdefiniujemy automat niedeterministyczny. Po udowodnieniu równoważności automatu deterministycznego i niedeterministycznego zajmiemy się poprawianiem efektywności automatów skończonych: zademonstrujemy algorytmy usuwania zbędnych przejść automatu skończonego, oraz pokażemy jak na bazie automatu skończonego zbudować równoważny mu skończony automat minimalny.*

2.1 Formalna definicja automatu skończonego

Automat skończony (*AS)* jest piątką: $A = (Q, \Sigma, \delta, q_0, F)$, gdzie:

- Q jest skończonym zbiorem stanów
- Σ jest alfabetem symboli wejściowych
- $q_0 \in Q$ jest stanem startowym
- $F \subset Q$ jest zbiorem stanów końcowych
- δ jest funkcją przejścia zdefiniowaną w następujący sposób:
  1. $\delta: Q \times \Sigma \rightarrow Q$ (Dla automatu deterministycznego)
  2. $\delta: Q \times \Sigma \rightarrow P(Q)$ (Dla automatu nie-deterministycznego)

  Intuicyjnie, jeśli automat A znajduje się w stanie p oraz napotyka symbol a, wówczas znajdzie się w stanie $q = \delta(p, a)$

Przykład:

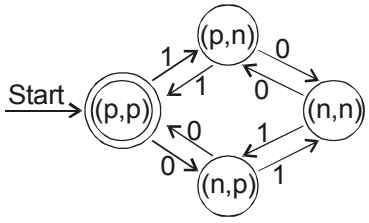
Weźmy alfabet binarny {0, 1}. Załóżmy, że mamy język składający się z wszystkich możliwych słów opartych na tym alfabecie i poszukujemy mechanizmu sprawdzającego czy dane na wejściu słowo posiada parzystą ilość zer i jedynek. Odpowiedzią może być zbudowanie następującego automatu deterministycznego: automat posiada 4 stany, odpowiadające 4 typom badanych słów (słowa mogą mieć parzystą / nieparzystą liczbę zer i jedynek). Stan będzie więc parą (x,y): $x,y \in \{p,n\}$, przy czym x określa czy liczba zer jest parzysta, natomiast y określa czy liczba jedynek jest parzysta. Formalna definicja tego automatu pokazana jest poniżej:

$A = (Q, \Sigma, \delta, q_0, F)$, gdzie
$Q=\{(p, p), (p, n), (n, p), (n, n)\}$
$\Sigma=\{0,1\}$
$q_0=(p, p)$,
$F=\{(p, p)\}$

Funkcja przejścia δ określona jest przez następującą tabelę (stan startowy poprzedzony jest symbolem $\rightarrow$, natomiast stan końcowy poprzedzony jest symbolem *):

		SYMBOLE	
		0	1
STANY	$\rightarrow$ * (p, p)	(n, p)	(p, n)
	(n, p)	(p, p)	(n, n)
	(p, n)	(n, n)	(p, p)
	(n, n)	(p, n)	(n, p)

Graficzna postać badanego automatu pokazana jest na rysunku nr 2.



Rysunek 2 Automat akceptujący słowa o parzystej liczbie 0 i 1.

2.2 Rozszerzenie tranzytywne funkcji δ

Intuicyjnie, automat skończony akceptuje słowa postaci $w = a_1a_2...a_n$ jeśli istnieje ścieżka w grafie automatu zaczynająca się w stanie startowym, kończąca w stanie końcowym i przechodząca przez łuki oznaczone kolejno symbolami a_1, a_2, ..., a_n. Formalnie rozszerza się funkcje przejścia δ do δ^* by działała zarówno na symbolach alfabetu jak i na całych słowach, tzn.:

- $q = \delta^*(p, w)$, jeśli startując ze stanu p w grafie automatu potrafimy dojść do stanu q odwiedzając kolejno łuki oznaczone symbolami tworzącymi słowo w.

Zatem: $\forall p \;\; p= \delta^*(p, \varepsilon)$ gdy ścieżka jest długości 0

Oraz: $\delta^*(p, wa) = \delta(\delta^*(p, w), a)$ gdzie w jest słowem, natomiast a jest pojedynczym symbolem alfabetu.

Łatwo dostrzec, że jeśli $w=a_1a_2...a_n$ oraz $\delta(p_i, a_i)=p_{i+1}$ dla każdego i=0,1,2,...,n to wówczas $\delta^*(p_0, w) = p_n$.

Akceptacja słów

Mówimy, że automat skończony $A = (Q, \Sigma, \delta, q_0, F)$ akceptuje słowo w, jeśli $\delta^*(q_0, w)$ jest stanem ze zbioru F.

Język automatu skończonego

Mówimy, że językiem automatu skończonego $A = (Q, \Sigma, \delta, q_0, F)$ jest zbiór:
$L(A) = \{w \mid \delta^*(q_0, w)$ jest stanem ze zbioru $F\}$

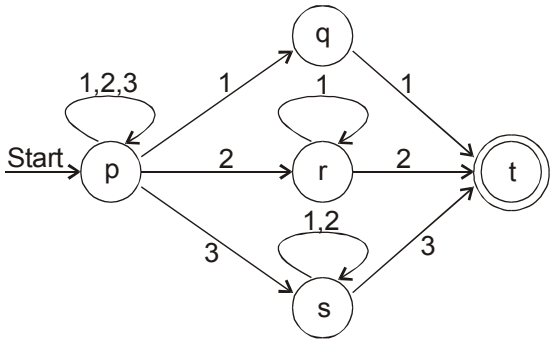
2.3 Niedeterministyczny automat skończony

Niedeterministyczny automat skończony to taki automat skończony, którego funkcja przejścia odwzorowuje stan na 0 lub więcej stanów pod wpływem pojedynczego symbolu alfabetu. Automaty niedeterministyczne są powszechnie stosowane przy konstrukcji kompilatorów, a ich niedeterministyczność jest tylko umowna, gdyż w rzeczywistości można je zawsze przekształcić do automatów deterministycznych.

Przykład:

Skonstruujemy teraz automat oparty na alfabecie {1,2,3}, który akceptuje tylko takie słowa, w których pomiędzy ostatnim symbolem słowa a wcześniejszym wystąpieniem tego *symbolu* są tylko symbole mniejsze, np.: .....11, ...2112, ...32211213.

Automat znajdujący się w stanie początkowym będzie akceptował dowolną ilość symboli 1, 2, 3 do czasu aż natknie się na przedostatnie wystąpienie ostatniego symbolu. Wówczas będzie próbował osiągnąć stan końcowy podążając jedną z trzech ścieżek w zależności od wartości symbolu końcowego. Sytuacja ta pokazana jest na rysunku nr 3.



Rysunek 3 Przykładowy automat niedeterministyczny

Formalna definicja niedeterministycznego automatu skończonego (NAS):

Niedeterministyczny automat skończony NAS jest podobnie jak deterministyczny automat skończony (DAS) piątką: $N = (Q, \Sigma, \delta, q_0, F)$, jednak funkcja przejścia δ zwraca zbiór stanów zamiast pojedynczego stanu.

Rozszerzenie tranzytywne funkcji δ dla NAS

Zachowanie się funkcji δ^* możemy zdefiniować indukcyjnie za pomocą następującego schematu indukcyjnego:

- *Baza*: $\delta^*(q, \varepsilon) = \{q\}$
- *Indukcja*: Niech $\delta^*(p, w) = \{q_1, q_2, ..., q_k\}$ oraz
  $\delta(q_i, a) = S_i$ dla i=1, 2, ..., k
  Wówczas $\delta^*(p, wa) = S_1 \cup S_2 \cup ... \cup S_k$

Język dla NAS

Niedeterministyczny automat skończony akceptuje słowo w jeśli jakakolwiek ścieżka od stanu startowego do stanu końcowego oznaczona jest przez w. Formalnie:
$L(A) = \{w \mid \delta^*(q_0, w) \cap F \neq \varnothing\}$

Równoważność NAS i DAS

To, że dla każdego automatu deterministycznego można stworzyć równoważny automat niedeterministyczny jest oczywistym faktem (wystarczy ograniczyć funkcje przejścia automatu niedeterministycznego do jednego stanu). Dowód w drugą stronę jest nieco trudniejszy.

Dla każdego niedeterministycznego automatu skończonego można stworzyć deterministyczny automat skończony. Tak stworzony automat skończony może jednak mieć eksponencjalny przyrost liczby stanów w stosunku do automatu wyjściowego.

Niech będzie dany dowolny NAS: $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$. Równoważnym dla niego DAS jest automat: $D = (Q_D, \Sigma, \delta_D, q_0, F_D)$, gdzie:

1. $Q_D = P(Q_N)$, czyli Q_D jest zbiorem wszystkich możliwych podzbiorów Q_N.
2. F_N jest zbiorem takich elementów S ze zbioru Q_D, że $S \cap F \neq \varnothing$.
3. δ_D jest zdefiniowana następująco:

$$\delta_D(\{p_1, p_2, ..., p_k\}, a) = \delta_N(p_1, a) \cup \delta_N(p_2, a) \cup ... \cup \delta_N(p_k, a)$$

Dla tak skonstruowanego automatu deterministycznego można udowodnić, że L(D) = L(N).

Przykład konstrukcji automatu równoważnego

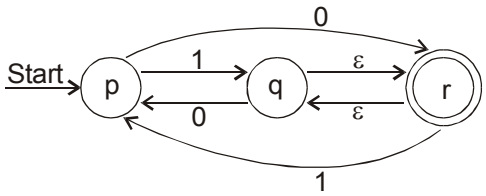
Dla automatu niedeterministycznego przedstawionego na rysunku nr 3 możemy stworzyć odpowiadający mu automat deterministyczny. Pierwotne podejście do tego zadania wymagałoby rozpatrywania 32 stanów automatu deterministycznego (tyleż jest podzbiorów 5 elementowego zbioru stanów automatu niedeterministycznego). W praktyce jednak można ograniczyć się jedynie do stanów osiągalnych. Przegląd automatu z rysunku nr 3 pozwala nam skonstruować funkcję przejścia automatu deterministycznego zawierającą 15 stanów osiągalnych:

	1	2	3
→p	pq	pr	ps
pq	pqt	pr	ps
*pqt	pqt	pr	ps
pr	pqr	prt	ps
*prt	pqr	prt	ps
ps	pqs	prs	pst
*pst	pqs	prs	pst
prs	pqrs	prst	pst
*prst	pqrs	prst	pst
pqs	pqst	prs	pst
*pqst	pqst	prs	pst
pqr	pqrt	prt	ps
*pqrt	pqrt	prt	ps
pqrs	pqrst	prst	pst
*pqrst	pqrst	prst	pst

2.4 Automat skończony i ε-eliminacja

Czasem może dojść do sytuacji, gdy graf reprezentujący automat będzie posiadał łuki oznaczone symbolem ε. W takim przypadku nic się nie zmienia; akceptacja słowa w będzie dalej uwarunkowana występowaniem ścieżki od stanu początkowego do stanu końcowego oznaczonej słowem w. Jeśli na tej ścieżce występuje symbol ε to nie ma on żadnego wpływu na słowo w.

Przykład



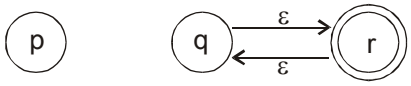
Rysunek 4 Automat przed ε-eliminacją

Automat z rysunku nr 4 akceptuje słowo 001 poprzez ścieżkę p→r→q→p→q→r dającą słowo 0ε01ε = 001.

Algorytm eliminacji przejść typu ε

Łuki typu ε są udogodnieniem; w żaden sposób nie wpływają one na możliwości automatów skończonych. W celu usunięcia łuków typu ε należy wykonać poniższe kroki:

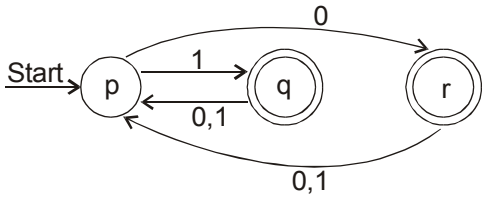
1. Wyznaczyć domknięcie tranzytywne wyłącznie łuków typu ε.
   - Przykład:



Domknięcie stanu p = {p}
Domknięcie stanu q = {q, r}
Domknięcie stanu r = {q, r}

2. Jeśli stan q jest osiągalny ze stanu p poprzez łuki typu ε, oraz jest łuk q→r oznaczony symbolem a różnym od ε, wówczas dodaj łuk p→r oznaczony symbolem a.
3. Uczyń ze stanu p stan akceptujący, jeśli jakiś stan akceptujący q jest osiągalny ze stanu p poprzez łuki typu ε.
4. Usuń wszystkie łuki typu ε.

Na rysunku nr 5 pokazany jest automat z rysunku nr 4 po wykonania algorytmu ε-eliminacji.



Rysunek 5 Automat po ε-eliminacji

2.5 Automat minimalny

Automat minimalny AM akceptuje ten sam język co A, jednak posiada minimalną ilość stanów. Co ciekawe, każdy deterministyczny automat skończony posiada unikalny (z dokładnością do nazw stanów) automat minimalny.

Rozróżnialność stanów

Powiemy, że dwa stany automatu p, q są *rozróżnialne*, jeśli istnieje takie słowo s, które pozwoli na przejście z tylko jednego ze stanów p, q do stanu akceptującego. Poniżej przedstawiony jest schemat indukcyjny pozwalający na wygenerowanie grup stanów nierozróżnialnych.

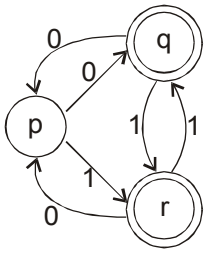
- *Baza*: dowolny stan akceptujący i nie-akceptujący są rozróżnialne. Możemy więc traktować bazę jako podział zbioru stanów na podzbiór stanów akceptujących i podzbiór stanów nie-akceptujących.
- *Indukcja*: p oraz q są rozróżnialne jeśli istnieje jakiś symbol a, dla którego $\delta(p,a)$ oraz $\delta(q,a)$ są rozróżnialne

Powyższy schemat pozwala odpowiedzieć na pytanie, czy dwa dowolne stany są rozróżnialne, czy nierozróżnialne. Pary nierozróżnialne mogą być łączone w większe grupy, gdyż nierozróżnialność jest tranzytywna.

Dowód tranzytywności dla nierozróżnialności jest następujący: załóżmy że p, q są nierozróżnialne, q, r są nierozróżnialne oraz (nie-wprost) p, r są rozróżnialne. Stąd istnieje takie słowo s, że $\delta(p, s)$ jest akceptujący, a $\delta(r, s)$ nie jest akceptujący. Jednak p, q są nierozróżnialne więc także $\delta(q, s)$ jest akceptujący, oraz q, r są nierozróżnialne więc $\delta(q, s)$ nie jest akceptujący – sprzeczność.

Przykład rozróżnialności

Przeanalizujmy rozróżnialność stanów automatu z rysunku nr 6.



Rysunek 6 Automat przed minimalizacją

- Z bazy indukcyjnej wiemy, że p jest rozróżnialny od q, r
- q, r są nierozróżnialne, ponieważ:
    - Żadne słowo zaczynające się od 0 nie rozróżni stanów q,r ponieważ symbol 0 spowoduje przejście ze stanów q,r do stanu p, z którego istnieje już jednoznaczna ścieżka.
    - Załóżmy, że słowo rozpoczyna się k-symbolami 1; Po tych k symbolach dalej będziemy się znajdować w stanach akceptujących q,r. Jeśli słowo się skończy, to q,r będą nierozróżnialne tym słowem. Jeśli natomiast po k symbolach 1 następuje symbol 0, to automat przejdzie do stanu p, skąd jest już dalej jednoznaczna ścieżka. Wniosek: q, r są nierozróżnialne.

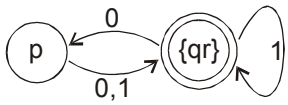
Konstrukcja minimalnego deterministycznego automatu skończonego

W celu skonstruowania minimalnego DAS $M=(Q_M, \Sigma, \delta_M, q_0, F_M)$ z danego automatu $A = (Q, \Sigma, \delta, q_0, F)$ należy:

1. Dla każdej grupy stanów nierozróżnialnych (w wyniku tranzytywności w grupie może być wiele stanów) wybrać reprezentanta tej grupy (jeden stan z grupy)
2. Za Q_M przyjąć reprezentantów grup stanów nierozróżnialnych. Za Q_M przyjąć reprezentantów grup stanów akceptujących.
3. Jeśli p jest reprezentantem grupy do której nie należy q oraz $\delta(p,a) = q$ w automacie A, wówczas należy dodać do automatu M łuk oznaczony symbolem a łączący p z reprezentantem grupy, do której należy q. Formalnie:$\delta_M(p,a)$=Reprezentant(Grupa(q)). Jeśli q jest jedynym elementem w swojej grupie, dodajemy łuk $\delta_M(p,a) = q$.
4. Usunąć wszystkie stany nieosiągalne ze stanu startowego.

Przykład:

Zminimalizowany automat z rysunku nr 6 pokazany jest na rysunku nr 7.



Rysunek 7 Automat zminimalizowany

Tak skonstruowany automat minimalny jest rzeczywiście minimalny

□

Mamy DAS A, który został zminimalizowaliśmy powyższym schematem do automatu M. Załóżmy nie-wprost, że istnieje automat DAS N, który akceptuje ten sam język co A, lecz posiada mniej stanów od M. Dowód sprzeczności tego założenia jest następujący:

- Badamy równocześnie na M i N równoważność stanów
- Stany startowe M oraz N są nierozróżnialne, gdyż jeśli jakieś słowo jest akceptowane przez M, to jest także akceptowane przez N (L(M) = L(N)). Podobnie jeśli jakieś słowo nie jest akceptowane przez M, to nie jest również akceptowane przez N.
- Jeśli jakieś dwa stany p,q są nierozróżnialne, to z definicji $\delta(p,a)$ oraz $\delta(q,a)$ też są nierozróżnialne dla dowolnego symbolu a.
- Stąd, skoro ani M ani N nie posiadają stanów nieosiągalnych, każdy stan M jest nierozróżnialny od co najmniej jednego stanu N.
- Jeśli N ma mniej stanów niż M, to w M można określić dwa stany, które są nierozróżnialne od tego samego stanu w N; wobec tego te dwa stany są nierozróżnialne względem siebie.
- Z drugiej strony M został skonstruowany tak, by żadne jego dwa stany nie były nierozróżnialne – sprzeczność z założeniem, że N jest najmniejszym automatem akceptującym język L(A) ■

W rzeczywistości można udowodnić, że każdy automat minimalny jest identyczny z dokładnością do nazw stanów) z automatem minimalnym wygenerowanym przy pomocy przedstawionego schematu.

3. Wyrażenia regularne

Streszczenie: *Wyrażenia regularne, których dotyczy ten rozdział są algebraicznym odpowiednikiem automatów skończonych i są często używane w różnych językach przy opisie wzorców dla tekstu. Po wprowadzeniu formalnej definicji wyrażeń regularnych i ich praw algebraicznych udowodnimy równoważność wyrażenia regularnego i aparatu skończonego. W dalszej części rozdziału udowodnimy lemat o pompowaniu dla języków regularnych, który stanowi ważne narzędzie przy rozstrzyganiu, czy dany język jest regularny, czyli czy istnieje akceptujący go automat skończony. Pod koniec rozdziału skupimy się na własnościach rodziny języków regularnych: udowodnimy twierdzenie o podstawieniu, które wraz z lematem o pompowaniu pomoże nam rozstrzygnąć problem przynależności, pustości oraz skończoności języków regularnych.*

3.1 Operatory i operandy

Jeśli E jest wyrażeniem regularnym, wówczas L(E) oznacza język, za którym stoi E. Wyrażenia regularne są budowane w następujący sposób:

- Operandem może być:
    1. Zmienna reprezentująca język
    2. Symbol reprezentujący samego siebie jako zbiór słów tzn. a oznacza język {a} (Formalnie L(a) = {a})
    3. Symbol ε reprezentujący język {ε}
    4. Symbol ∅ reprezentujący język pusty ∅

- Operatorem może być:
    1. Znak + reprezentujący sumę; $L(E+F) = L(E) \cup L(F)$
    2. Zestawienie dwóch operandów; nie jest ono zaznaczane żadnym dodatkowym znakiem i oznacza zwykłą konkatenację; L(EF) = L(E)L(F), przy czym konkatenację definiujemy następująco: $AB = \{ab \mid a \in A \wedge b \in B\}$
    3. Znak * reprezentujący domknięcie; $L(E^*) = (L(E))^*$, przy czym $L^* = \{\varepsilon\} \cup L \cup LL \cup LLL \cup ...$

- Nawiasy mogą być użyte w celu zmiany kolejności grupowania, która standardowo ma postać: * (największy priorytet), konkatenacja, suma (najmniejszy priorytet)

Przykłady

- L(0101010) = {0101010}
- L($10^*1 + 01^*0$) = {11,00,101,010,1001,0110,10001,01110,...}
- L((0 (0+1))*) = zbiór ciągów zero - jedynkowych, które posiadają parzystą liczbę symboli, oraz na każdej nieparzystej pozycji stoi 0.

3.2 Równoważność automatów skończonych i wyrażeń regularnych

W celu pokazania równoważności języków automatów skończonych i wyrażeń regularnych:

- Pokażemy że dla dowolnego języka generowanego przez wyrażenie regularne da się skonstruować NAS z łukami typu ε akceptujący ten język
- Pokażemy, że da języka akceptowanego przez dowolny DAS da się skonstruować wyrażenie regularne RE generujące ten język.

Języki akceptowane przez DAS, NAS, ε-NAS, RE będziemy nazywali *regularnymi*.

Konstrukcja ε-NAS na podstawie RE

Konstrukcja ε-NAS na podstawie RE jest stosunkowo prosta. Automat taki będzie posiadał jeden stan akceptujący. Konstrukcja jego stanów oparta będzie na poniższym schemacie:

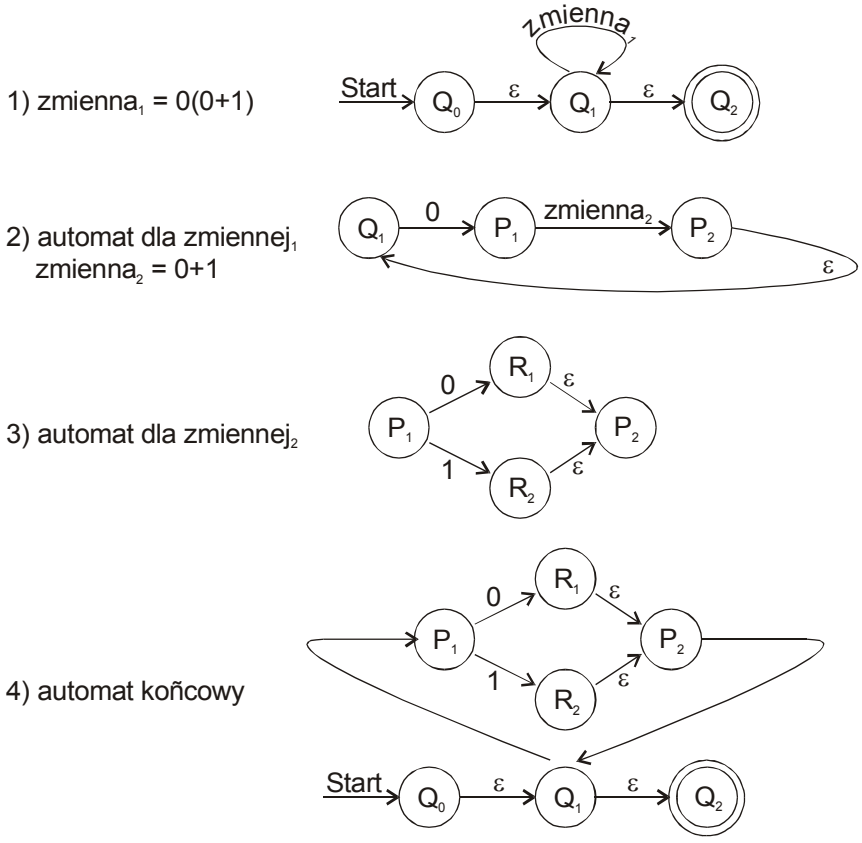
1. Podstaw za każde wyrażenie w nawiasie (...) jakąś zmienną Z_i (zmienne nie mogą być takie same). Za każdym operatorem * wstaw operand ε, np. $0^*+1 = 0^*\varepsilon+1$.
2. Teraz nasze wyrażenie regularne jest postaci $Z_1O_1Z_2O_2 \ldots O_{n-1}Z_n$, gdzie O_i są operatorami {konkatenacja, +, *}, natomiast Z_i są operandami bądź zmiennymi.
3. Stanem startowym ε-NAS będzie Q_0 natomiast akceptującym Q_{n+1}. Dodaj do automatu stany Q_1, Q_2, ..., Q_n odpowiadające kolejnym operatorom. Na razie stany te nie są połączone.
4. Dla każdej pary Z_iO_i gdzie O_i jest operatorem *, dodaj do stanu Q_i łuk $Q_i \rightarrow Q_i$ oznaczony symbolem Z_i.

5. Dla każdej sekwencji $Z_iO_iZ_{i+1}$ gdzie O_i jest konkatenacją połącz stany Q_i oraz Q_{i+1} łukiem oznaczonym symbolem Z_{i+1}.
6. Spójrz na RE pod kątem operatora + tzn. podziel go na podciągi rozdzielone operatorem +: $podciąg_1$ + $podziąg_2$ + ... + $podciąg_k$. Teraz dla każdego podciągu j postaci: $Z_rO_rZ_{r+1}O_{r+1}...O_{s-1}Z_s$ wykonaj (indeksy r oraz s są różne dla kolejnych j):
   a. Jeśli $O_r \neq *$, połącz Q_0 z Q_r symbolem Z_r; jeśli $O_r = *$ połącz Q_0 z Q_r symbolem ε.
   b. Połącz Z_s z Q_{n+1} symbolem ε.
7. Dla *każdego* łuku $Q_i \rightarrow Q_j$ oznaczonego zmienną Z_k, usuń ten łuk, podstaw za RE wyrażenie schowane w zmiennej Z_k i przejdź do kroku 1 algorytmu przyjmując nowe zmienne Z, O, Q, przy czym $Q_0 := Q_i$ oraz $O_{n+1} := Q_j$.

Działanie tego algorytmu dla wyrażenia regularnego $(0(0+1))^*$ pokazane jest na rysunku nr 8.

■

Rysunek 8 Automat dla wyrażenia regularnego (0(0+1))*



Konstrukcja RE na podstawie automatu skończonego

Dla konstrukcji RE na podstawie automatu skończonego wykorzystamy prosty schemat indukcyjny:

- Niech A będzie automatem skończonym ze stanami 1, 2, ..., n.
- Niech $R_{ij}^{(k)}$ będzie RE którego językiem jest zbiór słów generowanych przez ścieżki wychodzące ze stanu i, wchodzące do stanu j oraz *nie* przechodzące przez *żaden* stan o numerze większym od k.
- Dowód indukcyjny równoważności automatów i tak określonych wyrażeń regularnych konstruowany jest w zależności od wartości k.

Baza: k=0, Ścieżka nie może przechodzić przez żaden stan.

- Ścieżka jest zatem łukiem lub jest zerowa (pojedynczy stan)
- Jeśli $i \neq j$ wówczas $R_{ij}^{(0)}$ będzie sumą wszystkich takich symboli a, że automat A posiada łuk $i \rightarrow j$ oznaczony symbolem a.
- Jeśli i=j wówczas do zbioru $R_{ij}^{(0)}$ należy dodać symbol ε.

Indukcja: Załóżmy, że poprawnie określiliśmy wyrażenia dla zbiorów $R^{(k-1)}$. Zatem dla $R^{(k)}$ będziemy mieli:

- $R_{ij}^{(k)} = R_{ij}^{(k-1)} + R_{ik}^{(k-1)} \left(R_{kk}^{(k-1)} \right)^* R_{kj}^{(k-1)}$

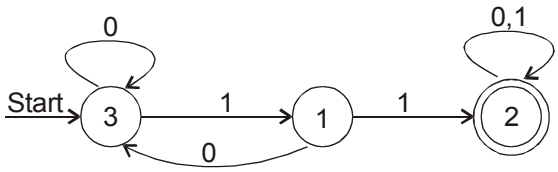
Uzasadnienie poprawności powyższej reguły jest następujące: ścieżka od i do j nie przechodząca przez stany większe od k:

1. Nigdy nie przechodzi przez stan k; W tym przypadku słowo określające tą ścieżkę musi (z założenia indukcyjnego) być generowane przez $R_{ij}^{(k-1)}$.
2. Przechodzi przez stan k jeden lub więcej razy; W takim przypadku:
   - $R_{ik}^{(k-1)}$ określa tą porcję ścieżki, która startując od stanu i dociera pierwszy raz do k
   - $\left(R_{kk}^{(k-1)} \right)^*$ określa tą porcję ścieżki (może być pusta!), która wielokrotnie wychodzi z k przechodzi przez stany nie większe od k-1 i wraca do k
   - $R_{kj}^{(k-1)}$ określa tą porcję ścieżki, która wychodzi po raz ostatni z k oraz dociera do j

Dla automatu skończonego A o n stanach, odpowiadające mu wyrażenie regularne RE jest sumą wyrażeń regularnych $R_{ij}^{(n)}$ gdzie i jest stanem początkowym A, natomiast j jest jednym ze stanów akceptujących A. ■

Przykład:

Rozpatrzmy przykładowy automat



Rysunek 9 RE generowane na podstawie automatu

- Obliczamy bazowe wyrażenie regularne (dla k=0):
  $R_{11}^{(0)} = \varepsilon$, $R_{12}^{(0)} = 1$, $R_{22}^{(0)} = \varepsilon + 0 + 1$, $R_{31}^{(0)} = 1$, $R_{32}^{(0)} = R_{21}^{(0)} = \varnothing$, $R_{33}^{(0)} = \varepsilon + 0$
- Obliczamy wyrażenia $R^{(k)}$ dla kolejnych k, jak np.:
  $R_{32}^{(1)} = R_{32}^{(0)} + R_{31}^{(0)} (R_{11}^{(0)})^* R_{12}^{(0)} = \varnothing + 1\varepsilon^*1 = 11$
  $R_{22}^{(1)} = R_{22}^{(0)} + R_{21}^{(0)} (R_{11}^{(0)})^* R_{12}^{(0)} = \varepsilon + 0 + 1 + \varnothing\varepsilon^*1 = \varepsilon + 0 + 1$
  gdyż $\varnothing R = R\varnothing = \varnothing$, ponieważ $\varnothing$ jest eliminatorem dla konkatenacji

3.3 Prawa algebraiczne dla wyrażeń regularnych

Rozszerzenie wyrażeń regularnych

Badając wyrażenia regularne często posługujemy się formami:

- E? oznacza 0 lub jedno wystąpienie E; Formalnie $E? = \varepsilon + E$
- E^+ oznacza 1 lub więcej wystąpień E; Formalnie $E^+ = EE^*$

Jeśli dwa wyrażenia regularne E, F nie posiadają zmiennych, wówczas E = F oznacza L(E) = L(F) (wyrażenia nie są identyczne, ale generują te same języki), przykładowo $1^+ = 11^*$.

Jeśli dwa wyrażenia regularne E, F posiadają zmienne, wówczas E = F (E jest równoważne F) oznacza, że podstawiając jakiekolwiek języki za zmienne w E i F, otrzymamy identyczne języki tworzone przez E i F. Przykładowo $E^+ = EE^*$.

Z wyjątkiem dwóch sytuacji (opisane poniżej), operator + możemy utożsamiać z dodawaniem, gdzie $\varnothing$ jest elementem neutralnym (0), natomiast operator konkatenacji możemy utożsamiać z mnożeniem, gdzie ε jest elementem neutralnym (1).

- Obydwa operatory: + oraz konkatenacja są łączne
- Operator + jest przemienny

- Prawa identyczności są prawdziwe dla + oraz konkatenacji
- ∅ jest eliminatorem (elementem zerującym) dla konkatenacji
- Wyjątki:
    - Konkatenacja nie jest przemienna: ab ≠ ba
    - Dla dowolnych E, operator + ma własność E + E = E.

3.4 Sprawdzanie praw dla wyrażeń regularnych

Sprawdzanie praw dla wyrażeń regularnych sprowadza się do odpowiedzi na pytania następującej treści: „Czy dla dowolnych wyrażeń R, S zachodzi: $(R + S)^* = (R^*S^*)^*$?".

Przy badaniu czy zachodzi równość dwóch wyrażeń regularnych, które posiadają „miejsca na języki" (jak R, S), zamiast konkretnych symboli (np. 1,2,3), strategia jest następująca: traktujemy R, S jako zwykłe symbole, np. R={0}, S={1}, a następnie badamy, czy $(0 + 1)^* = (0^*1^*)^*$.

- Wyrażenie po lewej stronie może wygenerować dowolny ciąg binarny, gdyż jest to generowanie w nieskończoność znaków 0 lub 1.
- Wyrażenie po prawej stronie może wygenerować dowolny ciąg binarny, gdyż jest to w szczególnym przypadku generowanie w nieskończoność ciągów 0 lub 1.

Jeśli przy sprawdzaniu jakiegoś prawa dla wyrażeń regularnych okaże się, że nie jest ono spełnione dla jakiegoś podstawienia symboli za języki, wówczas nie mamy do czynienia z prawem. Co ciekawe, jeśli okaże się, że dla jakiegoś podstawienia symbolów za wszystkie wystąpienia zmiennych w prawie okaże się, że dwa wyrażenie regularne są równe, wówczas dla *dowolnego* języka podstawionego za te same zmienne otrzymamy równość wyrażeń regularnych.

Jedynym problemem jest więc sprawdzenie, czy wyrażenie regularne zbudowane z samych symboli języka (bez zmiennych) są równe (produkują ten sam język).

3.5 Lemat o pompowaniu

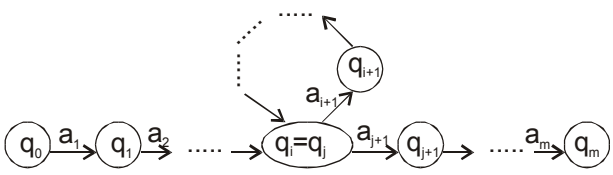
Lemat o pompowaniu to solidne narzędzie pozwalające udowodnić, że język nie jest regularny:

Jeśli L_1 jest językiem regularnym, wówczas istnieje taka stała n, że każde słowo w L_1 o długości n lub większej może być zapisane jako s=xyz, gdzie:

1. $0 < |y|$
2. $|xy| \leq n$
3. Dla każdego $k \geq 0$ xy^kz jest także w L_1 (y^k = y powtórzone k razy; $y^0 = \varepsilon$)

Dowód:□

- Skoro zakładamy, że L_1 jest językiem regularnym, to musi istnieć deterministyczny automat skończony A, taki że $L(A) = L_1$.
- Przyjmujemy za n – ilość stanów automatu A.
- Weźmy dowolne słowo z języka L_1 o długości $m \geq n$, np. $s = a_1a_2...a_m$
- Skoro automat A akceptuje L_1, to akceptuje także słowo s; Ścieżka tego słowa musi przebiegać przez m+1 stanów automatu A, począwszy od stanu startowego, aż do jakiegoś stanu akceptującego. Niech stanami tymi będą: $q_0, q_1, ..., q_m$, gdzie $q_{i+1}=\delta(q_i,a_{i+1})$ dla i = 0, 1, ...,m-1.
- Skoro automat A posiada n różnych stanów, wśród początkowych n stanów $q_0, q_1, ..., q_n$, muszą istnieć dwa takie same stany np. q_i oraz q_j, gdzie $0 \leq i < j \leq m$.
- Niech $x = a_1a_2...a_i$; $y= a_{i+1}a_{i+2}...a_j$; $z = a_{j+1}a_{j+2}...a_m$ (Rysunek nr 10)



Rysunek 10 Ilustracja lematu o pompowaniu

- Powtarzając pętlę opartą na stanach $q_i, q_{i+1}, ..., q_j$ zero, lub więcej razy otrzymujemy słowa postaci $xy^kz = a_1a_2...a_i(a_{i+1}a_{i+2}...a_j)^ka_{j+1}a_{j+2}...a_m$ akceptowane przez A, czyli należące do L_1. Co więcej, skoro i oraz j są mniejsze od n (ponieważ stany q_i, q_j znajdowały się wśród pierwszych n stanów), to $|xy| \leq n$. ■

Stosowanie lematu o pompowaniu

Poniższy schemat przedstawia jak zastosować lemat o pompowaniu by udowodnić, że język L nie jest regularny:

- Zakładamy, że L *jest* regularny
- Musi zatem istnieć jakaś stała n wykorzystywana w lemacie o pompowaniu. Możemy nie wiedzieć jakie jest n i traktować ją jako parametr.
- Wybieramy jakieś słowo s należące do L o długości większej od n.
- Stosując lemat o pompowaniu wiemy, że s może zostać rozbite na trzy części xyz spełniające założenia lematu. Tutaj także możemy nie wiedzieć w jaki sposób rozbić s, więc używamy x, y, z jako parametrów.
- Dochodzimy do sprzeczności wybierając takie i (które może zależeć od x, y, z, n), że dla żadnych wartości n, x, y, z, słowo xy^iz nie należy do języka L.

Przykład zastosowania lematu o pompowaniu

Załóżmy, że chcemy udowodnić, że język L= $\{0^i \mid$ i jest kwadratem liczby naturalniej$\}$ nie jest regularny. W tym celu:

- Twierdzimy, że L jest regularny, istnieje więc n spełniające założenia lematu o pompowaniu.
- Weźmy s = 0^{n^2}, które z pewnością jest w L. Z lematu o pompowaniu wiemy, że istnieje taki podział: s = xyz, dla którego $|xy| \leq n$ oraz $xyyz \in L$. Stąd długość słowa xyyz będzie większa od n^2, ale mniejsza bądź równa n^2+n.
- Następnym (pod względem długości) po s słowem w języku L jest słowo s'= $0^{(n+1)^2}$, które posiada $n^{2+}2n+1$ symboli 0. Skoro słowo xyyz jest dłuższe od s, oraz krótsze od s', nie może znajdować się w języku L $\Rightarrow$ sprzeczność.

Wniosek: Założenie, że L był językiem regularnym było błędne.

3.6 Własności rodziny języków regularnych

Pewne operacje na językach regularnych dają języki regularne, inne natomiast powodują powstawanie języków nieregularnych.

Podstawienie

Zdefiniujemy teraz operator postawienia. Załóżmy, że dany jest język regularny L oparty na alfabecie $\sum$. Niech będą dane dowolne języki regularne L_a dla każdego symbolu a z alfabetu $\sum$. Podstawieniem dla symbolu a będzie odpowiadający mu język regularny L_a. Formalnie: $p(a) = L_a$, dla każdego $a \in \sum$ (to tak, jakbyśmy z każdym symbolem alfabetu utożsamiali jakiś dowolny język regularny).

- Podstawienie rozszerzamy na słowa tzn. $p(a_1a_2...a_n) = p(a_1)p(a_2)...p(a_n) = L_{a1}L_{a2}...L_{an}$, co jest konkatenacją języków regularnych.
- Podstawienie rozszerzamy na języki tzn. $p(M) = \cup_{s \in M} p(s)$

Twierdzenie o podstawieniu: Dla dowolnego języka regularnego L opartego na alfabecie $\sum$ i dowolnych języków regularnych L_a dla każdego $a \in \sum$, język p(L) jest regularny

Dowód:□

Udowodnimy, że istnieje wyrażenie regularne E generujące p(L), czyli L(E)=p(L)

Niech R będzie wyrażeniem regularnym dla języka regularnego L. Ponadto, niech R_a będzie wyrażeniem regularnym dla każdego języka regularnego L_a podstawianego w miejsce symbolów a alfabetu ∑.

Konstrukcja poszukiwanego wyrażenia E jest następująca: wystartuj z R i podmień wszystkie symbole a wyrażeniem R_a.

Teraz wystarczy udowodnić, że dla tak skonstruowanego E zachodzi L(E) = p(L). Przeprowadzimy dowód indukcyjny ze względu na wielkość wyrażenia R

Baza: Jeśli R jest pojedynczym symbolem a, wówczas L={a}, E = R_a,

p(L) = p({a}) = L_a = L(R_a) = L(E)

- Przypadki gdy a jest ∅ lub ε są proste

Indukcja: Są trzy przypadki, w zależności czy $R=R_1+R_2$, $R=R_1R_2$, $R=R_1*$. Ograniczymy się tylko do przypadku $R=R_1R_2$. Skoro $R = R_1R_2$, to $L = L_1L_2$ gdzie $L_1=L(R_1)$ oraz $L_2=L(R_2)$. Niech E_1 oznacza R_1 z symbolami a zastąpionymi R_a (podobnie E_2). Z założenia indukcyjnego $L(E_1)=p(L_1)$ oraz $L(E_2)=p(L_2)$. Wobec tego $L(E) = L(E_1)L(E_2) = p(L_1)p(L_2) = p(L)$ ■

Zastosowanie twierdzenia o podstawieniu:

Stosując twierdzenie o podstawieniu można łatwo udowodnić, że jeśli L_1 oraz L_2 są regularne, to:

- L_1L_2 jest regularny: zastosuj podstawienia $p(a)=L_1$, $p(b)=L_2$ do języka regularnego {ab}; $p(\{ab\})=p(a)p(b)=L_1L_2$ jest regularny.
- $L_1 \cup L_2$ jest regularny: zastosuj podstawienia $p(a)=L_1$, $p(b)=L_2$ do języka regularnego {a, b}; $p(\{a, b\})=p(a) \cup p(b)= L_1 \cup L_2$ jest regularny.
- L_1^* jest regularny: zastosuj podstawienie $p(a) = L_1$ do języka $\{a^*\}$.

Przykład homomorfizmu

Niech $L = L(0^*1^*)$, a h będzie homomorfizmem zdefiniowanym w następujący sposób: h(0) = aa, h(1) = ε. Wówczas $h(0^*)=(aa)^*$, $h(1^*)=\varepsilon^*=\varepsilon$, $h(0^*1^*)=(aa)^*$. Homomorfizm dla języka definiujemy w następujący sposób:

h(L) = {h(s) | s∈ L}

czyli $h(\{0^*1^*\})= \{h(0^*1^*)\} = \{(aa)^*\} = L((aa)^*)$

Jeśli L jest regularny, to h(L) też jest regularny: automat B dla h(L) generujemy następująco:

- L jest regularny, więc istnieje automat A, który go rozpoznaje. Automat B posiada takie same stany jak automat A; Alfabet automatu B to h(alfabet automatu A). Funkcja przejścia B, to $\delta_B(p, h(a)) = q$ dla każdego a, takiego że $\delta_A(p, a) = q$.

Domknięcie przy inwersji homomorfizmu

Dla języka L, inwersja homomorfizmu h dla L ma następującą postać:
$h^{-1(}L) = \{s \mid h(s) \in L\}$

Jeśli L jest regularny, to $h^{-1(}L)$ też jest regularny: automat B dla $h^{-1}(L)$ generujemy podobnie jak w przypadku h(L), jednak tutaj $\delta_B(p, a) = q$ dla każdego h(a), takiego że $\delta_A(p, h(a)) = q$.

Domknięcie przy odwróceniu

Dla dowolnego słowa $s = a_1a_2...a_{n-1}a_n$, odwróceniem słowa jest $s_R=a_na_{n-1}...a_2a_1$. W szczególności $a^R = a$ dla pojedynczego symbolu a. Odwrócenie języka ma postać:
$L^R = \{s^R \mid s \in L\}$

Twierdzenie: Jeśli L jest regularny, to L^R też jest regularny
Dowód:□

Niech $A = (Q_A, \Sigma, \delta_A, q_{0A}, F_A)$ akceptuje L. Stworzymy NAS $B=(Q_B, \Sigma, \delta_B, start_B, F_B)$ zawierający ε-łuki akceptujący L^R:

- $Q_B = Q_A \cup \{start_B\}$, $F_B = \{q_{0A}\}$
- Dla każdego stanu $q \in F_A$ dodaj do B łuk $start_B \rightarrow q$ oznaczony symbolem ε
- Dla każdych p, q, a takich że $\delta_A(p, a) = q$ dodaj do B $\delta_B(q, a) = p$.

Jak łatwo dostrzec automat B generuje ścieżki odwrotne do ścieżek generowanych przez automat A; Stąd, automat B akceptuje wszystkie odwrócone słowa języka L. ■

3.7 Problemy decyzyjne dla języków regularnych

Mając daną reprezentację języka regularnego (automat skończony lub wyrażenie regularne) możemy odpowiedzieć na pewne pytania dotyczące tego języka. Prawidłowy wybór reprezentacji może ułatwić odpowiadanie na niektóre pytania.

Przynależność

Czy dane słowo s należy do języka L ?

1. Wybierz reprezentację L przez automat skończony
2. Przeprowadź symulację tego automatu dla ciągu wejściowego s. Jeśli automat osiągnie stan końcowy dla słowa s, to $s \in L$.

Pustość

Czy dany język L jest pusty: $L = \varnothing$?

1. Wybierz reprezentację L przez automat skończony
2. Użyj algorytmu przeglądu grafu począwszy od stanu startowego i sprawdź czy algorytm ten dotarł do jakiegokolwiek stanu końcowego. Jeśli tak – język nie jest pusty.

Skończoność

Czy dany język L jest skończony? Czy istnieje k, $|L| < k$?

Warto zauważyć, że dowolny język skończony jest regularny (możemy ponumerować jego słowa i z każdym słowem związać automat skończony, a następnie stany startowe tych automatów połączyć ze wspólnym stanem startowym łukiem ε). Z drugiej strony, nie każdy język regularny jest skończony np. $L=\{1^*\}$. Są dwie metody rozstrzygania czy język L jest skończony:

Metoda deterministycznego automatu skończonego

- Dla danego DAS akceptującego język L, wyeliminuj wszystkie stany, których nie da się osiągnąć ze stanu startowego. Następnie wyeliminuj stany, z których nie osiągnie się żadnego stanu końcowego.
- Sprawdź, czy w okrojonym automacie deterministycznym są cykle. Jeśli tak – język L jest nieskończony; Jeśli nie – język L jest skończony.

Metoda wyrażenia regularnego: W celu zbadania, czy język L reprezentowany przez wyrażenie regularne R jest skończony *prawie* wystarczyłoby sprawdzić czy w tym wyrażeniu występuje operator *: jeśli tak – język jest nieskończony; Jeśli nie – język jest skończony. Niemniej jednak istnieją pewne wyjątki dla tej zasady, które trzeba rozpatrzyć np.: $0\varepsilon^*1$ lub $0^*\varnothing$ są skończone. Algorytm sprawdzania czy język jest skończony składa się z 4 kroków:

1. Znajdź w R pod-wyrażenia równoważne z ∅; Wykorzystaj schemat indukcyjny:
   - *Baza*: ∅ jest równoważne ∅, Symbole a, ε nie są.
   - *Indukcja*: E+F jest równoważne $\varnothing \Leftrightarrow$ E i F są równoważne ∅
     EF jest równoważne $\varnothing \Leftrightarrow$ E lub F są równoważne ∅

E^* nigdy nie jest równoważne 0. Gdy E=∅, to E* = ε

2. Wyeliminuj pod-wyrażenia równoważne ∅:
   - Zamień na ∅ wszystkie wyrażenia E równoważne ∅.
   - Zamień E + F lub F + E na F gdy E jest równoważne ∅, a F nie.
   - Zamień E* na ε, gdy E jest równoważne ∅.

3. Znajdź w R pod-wyrażenia równoważne z ε; Wykorzystaj schemat indukcyjny
   - *Baza*: ε jest równoważny ε, Symbol a nie jest
   - *Indukcja*: E+F jest równoważne ε ⇔ E i F są równoważne ε
     EF jest równoważne ε ⇔ E i F są równoważne ε
     E^* jest równoważne ε ⇔ E jest równoważne ε.
4. Teraz, jeśli w okrojonym wyrażeniu R występuje pod-wyrażenie E^*, takie że E *nie* jest równoważne ε, to L(R) jest nieskończony.

Przykład:

Rozpatrujemy skończoność języka generowanego przez wyrażenie $(0 + 1\varnothing)^* + 1\varnothing^*$

Krok 1): ∅ (dwukrotnie) oraz 1∅ są pod-wyrażeniami równoważnymi ∅

Krok 2): Po eliminacji wyrażeń równoważnych ∅ pozostaje $0^* + 1\varepsilon$

Krok 3): Tylko pod-wyrażenie ε jest równoważne ε

Krok 4): W wyrażeniu występuje 0^*, a 0 nie jest równoważne ε więc L jest nieskończony

4. Podstawy języków bez-kontekstowych

Streszczenie: *Języki bez-kontekstowe stanowią kolejny po językach regularnych etap w hierarchii języków Chomsky'ego. Języki bez-kontekstowe pełnią w informatyce istotną rolę, gdyż można przy ich pomocy zapisywać składnię języków programowania tworząc tym samym podstawy dzisiejszych kompilatorów. Ten rozdział rozpoczyna się definicją języków bez-kontekstowych oraz akceptujących je automatów ze stosem wraz z przykładami ilustrującymi te pojęcia. W rozdziale pokazana zostanie równoważność drzew wywodów dla słowa wygenerowanego przez gramatykę bez-kontekstową, równoważność automatów zatrzymujących się przy pustym stosie z automatami zatrzymującymi się w stanach końcowych oraz równoważność języków generowanych przez gramatykę bez-kontekstową z językami akceptowanymi przez automat ze stosem.*

4.1 Gramatyki

Do tej pory analizowaliśmy języki regularne, które były tworzone przez automaty skończone lub wyrażenia regularne. Języki regularne stanowią najwęższą grupę języków w podziale klas języków zaproponowanym przez Chomsky'ego. Następną po regularnych klasą języków są języki bez-kontekstowe. Do definiowania języków bez-kontekstowych powszechnie stosuje się tzw. gramatyki. Gramatyka jest rekurencyjną definicją języka.

Przykład zapisu prostych tabel w języku HTML:

Start → <TABLE BORDER> Rows </TABLE>
Rows → Row Rows
Rows → ε
Row → <TR> Columns </TR>
Columns → Column Columns
Columns → ε
Column → <TD> Chars </TD>
Chars → a, b, ..., z ,A, B, …, Z, 0, 1, …, 9 (Dowolne znaki)
Chars → ε

Tak zdefiniowanej gramatyki:

- Słowem będzie np.:

<TABLE BORDER><TR><TD>Foka</TD> <TD>Ryba</TD></TR>
</TABLE>

Foka	Ryba

- *Zmiennymi* (inaczej symbolami *nie-terminalnymi*) będą zbiory słów, np. Rows, Columns, Chars
- Symbolami *terminalnymi* są symbole, z których składa się wygenerowany język, np. </TABLE>, <TD>, a, 7 itd. Mimo iż np. <TD> składa się z 4 znaków, możemy go utożsamiać z jednym terminalem.
- *Produkcjami* są wyrażenia postaci **Głowa** $\rightarrow$ **Ciało**, gdzie (dla gramatyk bez-kontekstowych) Głową jest jakiś *nie-terminal*, a Ciałem jest wyraz składający się z zera lub więcej *terminali* i/lub *nie-terminali*.
- Nie-terminal Start jest symbolem określającym początek języka.
- Notacja takiej gramatyki, to czwórka G = (V, $\sum$, P, S), gdzie V jest zbiorem symbolów nie-terminalnych, $\sum$ jest zbiorem symbolów terminalnych, P jest zbiorem produkcji, a S jest symbolem startowym ze zbioru nie-terminali.

Konstrukcja gramatyki z języka:

Wyobraźmy sobie język oparty na alfabecie binarnym {0,1} taki, że każde słowo tego języka składa się z kolejnych bloków zer i jedynek; przy czym każdy blok to pewna ilość zer, po której następuje co najmniej tyle samo jedynek. Język ten, mimo iż brzmi skomplikowanie jest generowany przez prostą gramatykę:

$S \rightarrow SA \mid \varepsilon$
$A \rightarrow 0A1 \mid A1 \mid 01$

Pionowa kreska oddziela różne ciała produkcji od tej samej głowy.

Wywody

Jeśli mamy słowo $s = \alpha A\beta$, oraz istnieje produkcja gramatyczna $A \rightarrow \gamma$, gdzie $\alpha,\beta,\gamma \in (V+\sum)^*$ oraz $A \in V$ to słowo s można przekształcić do $\alpha\gamma\beta$. W skrócie $\alpha A\beta \Rightarrow \alpha\gamma\beta$. Czasem zaznacza się również jaka gramatyka została użyta do wywodu: $\alpha A\beta \Rightarrow_G \alpha\gamma\beta$.

Tranzytywne domknięcie relacji $\Rightarrow$ oznaczamy symbolem $\Rightarrow^*$. Formalnie: $\alpha \Rightarrow^* \beta$ jeśli słowo α może zostać sprowadzone do słowa β w 0 lub więcej krokach wywodu.

- Przykładowo, 011AS $\Rightarrow^*$ 011AS (zero kroków wywodu); 011AS $\Rightarrow^*$ 0110A1S (jeden krok); 011AS $\Rightarrow^*$ 0110011 (trzy kroki).

Język gramatyk bez-kontekstowych (CFG)

Niech G = (V, $\sum$, P, S). Językiem generowanym przez gramatykę G jest zbiór wszystkich słów złożonych z symboli terminalnych, które mogą być wywiedzione z symbolu startowego S. Formalnie:

L(G)= {w: S$\Rightarrow^*$w, gdzie w$\in\sum^*$, S jest symbolem startowym gramatyki G}

Pomocnicza notacja

Przy zapisywaniu gramatyk istnieje „umowna" notacja: małymi symbolami alfabetu a,b,c,... oznaczamy symbole terminalne; x,y,z,... oznaczają wyrazy złożone z symboli terminalnych; literki greckie α,γ,β,... oznaczają wyrazy złożone zarówno z symboli terminalnych jak i nie-terminalnych; duże litery A, B, C,... oznaczają symbole nie-terminalne; symbole X, Y, Z,... oznaczają wyrazy złożone z symboli nie-terminalnych; S oznacza zazwyczaj symbol startowy gramatyki.

Wywody lewostronne i prawostronne

Przy każdym kroku wywodu mamy możliwość wyboru dla którego który symbol nie-terminalnego w słowie zastosujemy produkcję gramatyczną. W związku z tym wywody mogą się okazać różne tylko dlatego, że zastosowaliśmy różne porządki stosowania produkcji gramatycznych.

- Wywód lewostronny to taki wywód słowa, w którym za każdym razem zamieniony zostaje nie-terminal najbardziej po lewej stronie słowa. Przykładowo, dla słowa s=DCBD najpierw zastosowana zostanie produkcja gramatyczna dla D z lewej strony.
- Wywód prawostronny to taki wywód słowa, w którym za każdym razem zamieniony zostaje nie-terminal najbardziej po prawej stronie słowa. Przykładowo, dla słowa s=DCBD najpierw zastosowana zostanie produkcja gramatyczna dla D z prawej strony.

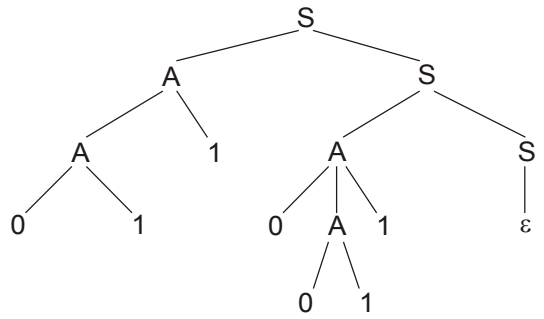
Wywód lewostronny oznacza się skrótowo symbolem $\Rightarrow_l$, a wywód prawostronny $\Rightarrow_p$.

Przykład wywodu lewostronnego i prawostronnego:

$S \Rightarrow_l AS \Rightarrow_l A1S \Rightarrow_l 011S \Rightarrow_l 011AS \Rightarrow_l 0110A1S \Rightarrow_l 0110011S \Rightarrow_l 0110011$
$S \Rightarrow_p AS \Rightarrow_p AAS \Rightarrow_p AA \Rightarrow_p A0A1 \Rightarrow_p A0011 \Rightarrow_p A10011 \Rightarrow_p 0110011$

Drzewa wywodów

Z każdym wywodem słowa w gramatyce bez-kontekstowej możemy związać drzewo wywodu. Węzły drzewa wywodu są terminalami, nie-terminalami lub symbolem ε. Każdy węzeł wewnętrzny drzewa (węzeł posiadający dzieci) musi być nie-terminalem. Terminale oraz symbole ε muszą być liśćmi drzewa. Jeśli symbol ε jest liściem drzewa, to jego rodzic nie może posiadać innych dzieci. Przykładowe drzewo wywodu jest pokazane na rysunku nr 11. Reprezentuje one wywód słowa 0110011.



Rysunek 11 Drzewo wywodu dla gramatyki bez-kontekstowej

Równoważność drzew wywodów i wywodów lewostronnych / prawostronnych

Dla gramatyki G = (V, ∑, P, S) i słowa terminalnego w następujące warunki są równoważne:

1. $S \Rightarrow^* w$ (czyli $w \in L(G)$)
2. $S \Rightarrow_l^* w$
3. $S \Rightarrow_p^* w$
4. Istnieje dla gramatyki G drzewo wywodu z korzeniem w S oraz liśćmi, których etykiety przeglądane od lewej do prawej tworzą słowo w.

$2 \Rightarrow 1$, $3 \Rightarrow 1$ są oczywiste

Dowód $1 \Rightarrow 4$ □

Przeprowadzimy indukcyjny ze względu na długość wywodu słowa w.

- *Baza*: Jeśli wywód ma długość 1, to drzewo wywodu istnieje i jest oczywiste.

- *Indukcja*: Załóżmy, że dla każdego słowa terminalnego w', którego długość wywodu jest mniejsza od k istnieje drzewo wywodu. Rozpatrzmy słowo w, którego długość wywodu jest równa k. Niech pierwszym krokiem wywodu w będzie $S \Rightarrow \alpha_1\alpha_2 ... \alpha_n$. Widzimy, że wywody słów α_1, α_2, ... α_n muszą być krótsze od k (bo inaczej wywód w byłby dłuższy od k). Stąd (z założenia indukcyjnego) słowa α_1, α_2, ... α_n posiadają drzewa wywodów T_1, T_2, ..., T_n. Przeglądając od lewej do prawej liście drzewa T_i otrzymamy słowo α_i dla i=1,2, ..., n. Jeśli więc połączymy drzewa T_1, T_2, ..., T_n we wspólne drzewo o nowym korzeniu S, gdzie T_1, T_2, ..., T_n będą kolejnymi (od lewej po prawej) dziećmi S, to S będzie drzewem wywodu słowa w.
    - Przykład: Rozpatrzmy wywód $S \Rightarrow AS \Rightarrow AAS \Rightarrow AA \Rightarrow A1A \Rightarrow A10A1 \Rightarrow 0110A1 \Rightarrow 0110011$. Poddrzewo wywodu A (krok 1) reprezentuje wywód $A \Rightarrow A1 \Rightarrow 011$. Poddrzewo wywodu S (krok 1) reprezentuje wywód $S \Rightarrow AS \Rightarrow A \Rightarrow 0A1 \Rightarrow 0011$. Łącząc te drzewa we wspólne drzewo o korzeniu S otrzymamy drzewo reprezentujące wywód słowa 0110011. ■

Dowód $4 \Rightarrow 2$ (dowód $4 \Rightarrow 2$ jest analogiczny) □

Przeprowadzimy indukcyjny ze względu na wysokość drzewa wywodu.

- *Baza*: Jeśli drzewo A ma wysokość 1, to liśćmi tego drzewa muszą być symbole terminalne a_1, a_2, ..., a_n przyczepione od lewej do prawej do korzenia A. Skoro drzewo wywodu reprezentuje zastosowanie produkcji gramatycznych, więc musi istnieć produkcja $A \rightarrow w$, gdzie $w = a_1a_2...a_n$. Stąd $A \Rightarrow_l w$ (bo słowo A posiada składa się z pojedynczego nie-terminalu A).
- *Indukcja*: Zakładamy, że mamy drzewo A o wysokości k, oraz że dla każdego drzewa o wysokości k-1 istnieje wywód lewostronny, który utworzy słowo składające się z liści drzewa o wysokości k-1 przeglądanych od lewej do prawej. Niech korzeń drzewa A ma dzieci X_1, X_2, ...X_m. Te X_i są korzeniami drzew o wysokości < k. Stąd (założenie indukcyjne) istnieją dla nich wywody lewostronne. Wywód lewostronny słowa w (złożonego z liści A przeglądanych od lewej do prawej) tworzymy następująco: Stosujemy najpierw produkcję $A \Rightarrow_l X_1X_2...X_m$, a następnie wywody lewostronne dla kolejnych pod-drzew X_1, X_2, ...X_m. Tak skonstruowany wywód lewostronny da nam w rezultacie słowo w. ■

Gramatyki niejednoznaczne

Gramatyka bez-kontekstowa jest niejednoznaczna, jeśli istnieje słowo terminalne generowane przez tą gramatykę, które posiada więcej niż jeden

różnych wywodów lewostronnych (równoważnie: wywodów prawostronnych lub drzew wywodów).

Przykład

Gramatyka, która została już przedstawiona tzn. $S \rightarrow AS \mid \varepsilon$; $A \rightarrow A1 \mid 0A1 \mid 01$ jest niejednoznaczna. Istotnie, wyraz 00111 posiada 2 różne wywody lewostronne (stosujemy w różnej kolejności produkcje $A \rightarrow A1$ oraz 0A1) :

1. $S \Rightarrow_l AS \Rightarrow_l 0A1S \Rightarrow_l 0A11S \Rightarrow_l 00111S \Rightarrow_l 00111$
2. $S \Rightarrow_l AS \Rightarrow_l A1S \Rightarrow_l 0A11S \Rightarrow_l 00111S \Rightarrow_l 00111$

Języki niejednoznaczne

Język jest niejednoznaczny jeśli każda generująca go gramatyka jest niejednoznaczna.

Przykład

Język generowany przez gramatykę $S \rightarrow AS \mid \varepsilon$; $A \rightarrow A1 \mid 0A1 \mid 01$ nie jest niejednoznaczny, gdyż można tą gramatykę zmienić, by była jednoznaczna i dalej generowała ten sam język. Pokazana poniżej gramatyka generuje „dodatkowe" jedynki na końcu:

$S \rightarrow AS \mid \varepsilon$

$A \rightarrow 0A1 \mid B$

$B \rightarrow B1 \mid B$

Pojęcie języków i gramatyk niejednoznacznych jest bardzo istotne. Niejednoznaczność gramatyki pociąga za sobą fakt, że co najmniej jedno generowane przez nią słowo ma różną strukturę wywodu (drzewo wywodu).

- Gramatyka niejednoznaczna nie jest dobrym kandydatem do reprezentacji języka programowania, ponieważ dwie różne struktury wywodu słowa (programu napisanego w tym języku) pociągałyby za sobą różne znaczenia tych programów.
- Przykładowo: proste gramatyki reprezentujące wyrażenia algebraiczne są niejednoznaczne i przy projektowaniu kompilatorów muszą zostać zastąpione bardziej skomplikowanymi gramatykami jednoznacznymi.

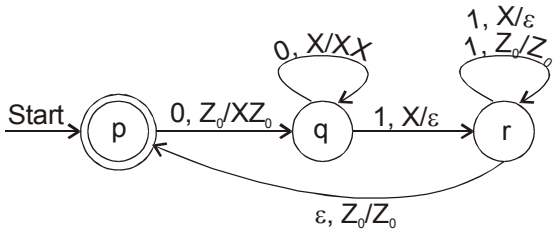
4.2 Automat ze stosem

Automaty ze stosem są przeważnie niedeterministyczne. Powstają one z automatów skończonych poprzez dodanie do automatu stosu opartego na jakimś alfabecie. Automaty ze stosem służą do rozpoznawania języków bez-kontekstowych.

Przykład

Zbudujemy teraz automat akceptujący język $S \rightarrow AS \mid \varepsilon$; $A \rightarrow A1 \mid 0A1 \mid 01$. Każdy łuk diagramu przejść pokazanego na rysunku nr 12 jest opisany w następujący sposób:

a, $Z / X_1X_2...X_n$, gdzie a jest aktualnym symbolem na wejściu automatu, Z jest symbolem na szczycie stosu automatu, $X_1X_2...X_n$ są kolejnymi symbolami, które zastąpią Z na stosie (X_1 będzie umieszczone na szczycie stosu).



Rysunek 12 Automat ze stosem

W stanach p i q automat akceptuje zera każdorazowo dodając na stos symbol X. Gdy na wejściu automatu pojawi się symbol 1 automat przechodzi do fazy dodawania jedynek. Faza dodawania jedynek trwa tak długo dopóki na stosie jest jakiś X (przy każdej dodanej jedynce ze szczytu stosu znika jeden X). Po wczytaniu tylu jedynek ile jest zer automat może kontynuować wczytywanie jedynek, albo przejść do stanu końcowego (także początkowego). Teraz automat może zakończyć wczytywanie słowa lub zacząć wczytywać kolejny blok zer i jedynek.

Formalna definicja automatu ze stosem

Automat ze stosem to siódemka: $A_S=(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, gdzie Q, Σ, q_0, F są takie same jak przy automatach skończonych, natomiast:

- Γ jest alfabetem stosu
- $Z_0 \in \Gamma$ jest symbolem startowym stosu. Na starcie na stosie znajduje się tylko Z_0
- δ jest funkcją przejścia, która bierze aktualny stan, symbol wejściowy (albo ε) oraz element na szczycie stosu i zwraca skończoną ilość par: (q, w_s) gdzie:
    - $q \in Q$ jest nowym stanem automatu (może to być stary stan)
    - $w_s \in \Gamma^*$ jest ciągiem symboli stosu, które zamienią aktualny szczyt stosu.

Jak widać, w przypadku automatu skończonego interesujący był tylko stan, w którym automat się znajdował. W przypadku automatu ze stosem interesuje nas zarówno aktualny stan automatu, jak i zawartość jego stosu (nie

tylko symbolu na szczycie). Dla automatu ze stosem w aktualnym momencie istotny jest więc *opis stanu* (q, w, α), gdzie q jest aktualnym stanem, w jest ciągiem symboli, które pozostały do zaakceptowania, natomiast α jest zawartością stosu (szczyt stosu to pierwszy symbol ciągu α).

Funkcja przejścia automatu ze stosem

Jeśli w wyniku przejścia $\delta(q, a, X)$ otrzymamy parę (p, α), to oznaczamy to w następujący sposób (używając relacji przejścia $\neg$) : $(q, aw, X\beta) \neg (p, w, \alpha\beta)$.

- Relacja $\neg$ jest tranzytywna, tzn. $\neg^*$ oznacza zero, jeden lub więcej przejść.
- Słowo wejściowe w jest akceptowane jeśli $(q_0, w, Z_0) \neg (p, \varepsilon, \beta)$ dla dowolnego stanu akceptującego p oraz dowolnego ciągu symboli stosowych β.
- $L(A_S) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \neg (p, \varepsilon, \beta) \text{ oraz } p \in F \}$

Przykład

Pokażemy, że automat z rysunku nr 12 akceptuje słowo 0101111:
$(p,0101111,Z_0) \neg (q,101111,XZ_0) \neg (r,01111,Z_0) \neg (p,01111,Z_0) \neg (q,1111,XZ_0) \neg (r,1111,Z_0) \neg (r,111,Z_0) \neg (r,11,Z_0) \neg (r,1,Z_0) \neg (r,\varepsilon,Z_0) \neg (p,\varepsilon,Z_0)$

Akceptacja przez pusty stos

Akceptacja przez pusty stos jest istotnym udogodnieniem. Pozwala nam ona twierdzić, że jeśli słowo jest akceptowane przez A_S to po przeczytaniu tego słowa stos automatu A_S jest pusty. Język akceptowany przez A_S przy akceptacji przez pusty stos jest zbiorem:

- $L_Z(A_S) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \neg (p, \varepsilon, \varepsilon) \}$

Przy akceptowaniu przez pusty stos automat nie musi mieć określonych stanów akceptujących.

Przykład

Automat z rysunku nr 12 można przekształcić do automatu, który będzie akceptował ten sam język przez pusty stos. Należy w tym celu dodać nowy element do funkcji przejścia:

- $\delta(p, \varepsilon, Z_0) = \{p, \varepsilon\}$

Powyższa modyfikacja funkcji przejścia pozwala automatowi zdjąć ze stosu pierwszy element (Z_0)zamiast rozpoczynać rozpoznawanie kolejnego

bloku zer i jedynek. W rezultacie p przestaje być stanem akceptującym. Co więcej, nie ma żadnego stanu akceptującego, gdyż akceptacja następuje przy pustym stosie.

Równoważność akceptacji przez stan końcowy i pusty stos

Istnieje automat ze stosem A_1 rozpoznający język $L=L(A_1)$ wtedy i tylko wtedy gdy istnieje automat ze stosem A_2 rozpoznający język $L=L_Z(A_2)$.

□

- Mając dany $A_1=(Q, \sum, \Gamma, \delta, q_0, Z_0, F)$, konstruujemy A_2:
    1. Rozpocznij od automatu A_2 identycznego z A_1.
    2. Dodaj do zbioru stanów automatu A_2 symbol p_0. Uczyń z p_0 stan startowy automatu A_2. Dodaj do alfabetu stosu automatu A_2 nowy symbol X_0. Uczyń X_0 startowym elementem stosu dla A_2.
    3. Dodaj do funkcji przejścia δ automatu A_2 element $\delta(p_0,\varepsilon,X_0) = (q_0,Z_0X_0)$. Wymusi to pierwszy ruch automatu A_2; obecność X_0 na dole stosu zapobiegnie „przypadkowej" akceptacji słowa gdy A_2 miałby pusty stos.
    4. Począwszy od stanu q_0, automat A_2 zachowuje się dokładnie tak jak automat A_1.
    5. Dodaj nowy stan e do zbioru stanów automatu A_2. W stanie tym automat A_2 będzie opróżniał swój stos, należy więc dodać do funkcji przejścia δ automatu A_2 elementy $\delta(e,\varepsilon,Z) = (e, \varepsilon, \varepsilon)$ dla każdego $Z\in\Gamma$.
    6. Jeśli symulowany przez A_2 automat A_1 znajduje się w stanie akceptującym dodaj przejście symbolem ε do stanu e. Formalnie dodaj do funkcji przejścia δ automatu A_2 elementy $\delta(f,\varepsilon,Z) = (e, \varepsilon, \varepsilon)$ dla każdego $Z\in\Gamma$ oraz $f\in F$. Daje to automatowi A2 możliwość opróżnienia swojego stosu bez wczytywania żadnych dodatkowych symboli.

- Mając dany $A_2=(Q, \sum, \Gamma, \delta, q_0, Z_0, F)$, konstruujemy A_1:
    1. Rozpocznij od automatu A_1 identycznego z A_2.
    2. Dodaj do zbioru stanów automatu A_1 symbol p_0. Uczyń z p_0 stan startowy automatu A_1. Dodaj do alfabetu stosu automatu A_1 nowy symbol X_0. Uczyń X_0 startowym elementem stosu dla A_1.
    3. Dodaj do funkcji przejścia δ automatu A_1 element $\delta(p_0,\varepsilon,X_0) = (q_0,Z_0X_0)$. Wymusi to pierwszy ruch automatu A_1; Jeśli automat A_1 zobaczy ponownie X_0 na szczycie stosu będzie to oznaczało, że aktualnie wczytany ciąg jest akceptowany przez A_2.

4. Dodaj nowy stan e do zbioru stanów automatu A_1. Będzie to jedyny stan akceptujący automatu A_1. Dodaj e do zbioru stanów akceptujących F.
5. Jeśli symulowany przez A_1 automat A_2 zobaczy na szczycie stosu symbol X_0, musi przejść do stanu e symbolem pustym ε. Formalnie dodaj do funkcji przejścia δ automatu A_1 elementy $\delta(q,\varepsilon,X_0) = (e, \varepsilon, \varepsilon)$ dla każdego $q \in Q-\{p_0\}$.

■

4.3 Równoważność gramatyk bez-kontekstowych i automatów ze stosem

Udowodnimy teraz następujące twierdzenie:
Język L jest bez-kontekstowy wtedy i tylko wtedy, gdy istnieje automat ze stosem go rozpoznający.

Dowód ⇒

Niech L będzie językiem bez-kontekstowym rozpoznawanym przez gramatykę $G=(V,\Sigma,P,S)$. Z wcześniejszego twierdzenia wiemy możliwe jest by każde słowo miało w tej gramatyce wywód lewostronny. Startując od S, każdy etap wywodu słowa w można zatem zapisać jako: xy, gdzie x jest słowem złożonym tylko z terminali, a y słowem złożonym tylko z nie-terminali. Konstruowana przez nas automat będzie operował głownie na swoim stosie, który będzie przechowywał fragment słowa pozostający do zaakceptowania. Jeśli na szczycie stosu będzie symbol terminalny, to automat będzie musiał napotkać ten symbol by przejść „dalej". Jeśli na szczycie stosu będzie nie-terminal, to automat będzie próbował zastąpić ten nie-terminal innym wyrażeniem (odpowiadającym produkcji gramatycznej, której głową jest ten nie-terminal). Formalna definicja jednostanowego automatu ze stosem akceptującego L przez pusty stos jest następująca:

$A=(\{q\},\Sigma,V\cup\Sigma,\delta,q,S)$, gdzie δ jest zdefiniowana jako:

1. Jeśli na szczycie stosu jest nie-terminal B, wówczas $\delta(q,\varepsilon,B)=\{(q,\alpha) \mid B\rightarrow\alpha$ jest w P$\}$
2. Jeśli na szczycie stosu jest terminal a, wówczas $\delta(q,a,a)=\{(q, \varepsilon)\}$

Jak widać, jeśli na szczycie stosu stoi nie-terminal B oraz istnieje produkcja gramatyczna zamieniająca B na α, to automat może „pustym symbolem" przejść do nowego stanu zdejmując ze stosu B i wrzucając na stos α (na szczycie stosu będzie pierwszy symbol słowa α). Jeśli natomiast na stosie

jest terminal a, to automat nie ma wyjścia, tylko oczekiwać, aż na wejściu pojawi się a. W każdym innym przypadku słowo nie zostanie zaakceptowane.

Przykład: Rozpatrzymy gramatykę: G=({S,A},{0,1},P,S) gdzie P składa się z następujących produkcji: $S \rightarrow AS \mid \varepsilon$; $A \rightarrow 0A1 \mid A1 \mid 01$:

A = ({q},{0,1},{0,1,A,S},δ,q,S), gdzie δ ma następującą postać:

- $\delta(q,\varepsilon,S)=\{(q, \varepsilon),(q,AS)\}$
- $\delta(q,\varepsilon,A)=\{(q, 0A1),(q,A1),(q,01)\}$
- $\delta(q,0,0)=\{(q, \varepsilon)\}$
- $\delta(q,1,1)=\{(q, \varepsilon)\}$

Dowód poprawność konstrukcji automatu: $L(G) = L_Z(A)$. Równość ta może być udowodniona gdy wykażemy, że każdej chwili wywodu w gramatyce G odpowiada poprawny układ automatu A. Formalnie, $S \Rightarrow_l^* xy$ wtedy i tylko wtedy gdy $(q,x\alpha,S) \neg^* (q, \alpha,y)$ gdzie x, α są dowolnymi ciągami terminali, a y jest dowolnym ciągiem nie-terminali.

Dowód indukcyjny $[S \Rightarrow_l^* xy] \Rightarrow [(q,x\alpha,S) \neg^* (q, \alpha,y)]$ ze względu na długość wywodu w gramatyce:

- *Baza*: Wywód długości 1, czyli $S \Rightarrow xy$. Automat początkowo zamienia na stosie S na xy, następnie akceptuje wszystkie x terminali usuwając je ze stosu by na stosie pozostały tylko symbole nie-terminalne y. Formalnie dla dowolnego α na wejściu automatu (następującego po x) mamy:
  $(q,x\alpha,S) \neg (q, x\alpha,xy) \neg^* (q, \alpha,y)$
- *Indukcja*: Niech założenie indukcyjne jest spełnione dla wywodów długości mniejszej od k. Rozbijmy wywód długości k: $S \Rightarrow_l^* x'y' \Rightarrow xy$, gdzie x=x't; y'=ny oraz $n \rightarrow t$ należy do P (x' jest prefiksem x, y jest sufiksem y'). Mamy zatem (korzystając z założenia indukcyjnego):
  $(q,x\alpha,S) = (q,x't\alpha,S) \neg (q, x't\alpha,x'y') \neg^* (q, t\alpha,y') = (q, t\alpha,ny) \neg (q, t\alpha,ty) \neg^* (q, \alpha,y)$ c.b.d.u.

Dowód indukcyjny $[(q,x\alpha,S) \neg^* (q, \alpha,y)] \Rightarrow [S \Rightarrow_l^* xy]$ ze względu na ilość przejść automatu A.

- *Baza*: Automat wykonał jedno przejście, czyli (S musi być nie-terminalem) automat wykonał przejście $(q,\alpha,S) \neg (q,\alpha,y)$ odpowiadające produkcji gramatycznej $S \rightarrow y$ ($x=\varepsilon$). Zatem prawdziwe jest $S \Rightarrow^* xy$
- *Indukcja*: Niech założenie indukcyjne jest spełnione jeśli automat wykona k przejść. Rozpatrzmy sytuację, gdy automat wykona k+1 przejść: $(q,x\alpha,S) \neg^* (q,\alpha,y) \neg (q, \alpha',y')$. Pierwsze k przejść daje wywód $S \Rightarrow_l^* xy$ (z założenia indukcyjnego). Teraz jeśli ostatnim przejściem automatu była akceptacja terminala t takiego, że $\alpha = t\alpha'$, wówczas na stosie musiała być produkcja postaci $n \rightarrow t$, przy czym $ny' = y$. Stąd jeśli przyjmiemy $x'=xt$

wówczas $S \Rightarrow_l^* xy = xny' \Rightarrow_l xty' = x'y'$. Jeśli natomiast ostatnim przejściem automatu była zamiana na stosie pierwszego symbolu z y na y', wówczas $\alpha=\alpha' y = ny_2$ oraz musi istnieć w P produkcja $n \rightarrow y'$. Wtedy jednak $S \Rightarrow_l^* xy=xny_2 \Rightarrow xy'y_2$, gdzie $y'y_2$ jest stosem automatu po k+1 przejściach c.b.d.u.

Konsekwencją dowodu $[(q,x\alpha,S) \neg^* (q, \alpha,y)] \Leftrightarrow [S \Rightarrow_l^* xy]$ jest fakt, że $[(q,y,S) \neg^* (q, \varepsilon,\varepsilon)] \Leftrightarrow [S \Rightarrow_l^* x]$ dla dowolnego słowa terminalnego x. W rezultacie dowolne słowo wygenerowane przez G jest akceptowane przez $L_Z(A)$, czyli $L(G) = L_Z(A)$.

■

Dowód $\Rightarrow$

Teraz zakładamy, że mamy automat za stosem $A=(Q,\Sigma,\Gamma,\delta,q_0,Z_0)$ akceptujący język L poprzez pusty stos: $L_Z(A) = L$. Skonstruujemy gramatykę bez-kontekstową G, która będzie akceptowała ten sam język. Idea konstrukcji będzie się opierała na wprowadzeniu do gramatyki G nie-terminali o następującej formie: [qZp], gdzie q, p są stanami automatu A, a Z jest symbolem na stosie automatu A.

Nie-terminal [qZp] będzie docelowo generował wszystkie takie ciągi terminalne, które zostaną zaakceptowane przez automat A na drodze od stanu q do p, przy jednoczesnej zmianie (netto) na stosie Z na ε. Formalnie:

- $[qZp] = \{w \in \Sigma^* \mid (q,wx,Z\alpha) \neg^* (p,x,\alpha)$ dla dowolnego $x \in \Sigma^*$ oraz $\alpha \in \Gamma^*\}$
- W szczególnym przypadku, gdy Z jest jedynym symbolem na stosie, a w jest całym ciągiem na wejściu automatu: $\{w \in \Sigma^* \mid (q,w,Z) \neg^* (p,\varepsilon, \varepsilon)\} \subset [qZp]$

Przykłady kilku nieterminali [qZp]:

- Jeśli mamy regułę akceptowania terminala przy zdjęciu ze stosu: $(p,\varepsilon) \in \delta(q,a,Z)$, to dodajemy następującą produkcję do gramatyki G: $[qZp] \rightarrow a$.
- Jeśli mamy regułę akceptowania terminala przy zamianie jednego symbolu na stosie na inny: $(p,Y) \in \delta(q,a,Z)$, to dla wszystkich stanów r dodajemy następującą produkcję do gramatyki G: $[qZr] \rightarrow a\,[pYr]$.
  Uzasadnienie: [qZr] mówi, że startujemy z q, mamy na stosie Z i zdążamy do r. Skoro $(p,Y) \in \delta(q,a,Z)$ to możemy zaakceptować symbol a, lecz wtedy znajdziemy się w stanie p, a na stosie zamiast Z będzie się znajdował Y. By znaleźć się teraz w stanie r przy pustym stosie, należy przejść z p do r i zdjąć ze stosu Y; w skrócie [pYr]. Ostatecznie więc mając za zadanie [qZr] i akceptując a, dostajemy nowe zadanie [pYr]. Stąd dodajemy do G produkcję $[qZr] \rightarrow a\,[pYr]$.

- Jeśli mamy regułę akceptowania terminala przy zamianie symbolu Z na szczycie stosu na dwa symbole XY: $(p,XY)\in\delta(q,a,Z)$, to dla wszystkich możliwych stanów r i s dodajemy następującą produkcję do gramatyki G: [qZs] $\rightarrow$ a [pXr] [rYs]. Uzasadnienie jest podobne do poprzedniego przypadku.

Formalna definicja gramatyki G, takiej że $L(G) = L_Z(A)$ jest następująca: Połóżmy $G=([\sum \Gamma \sum]\cup\{S\}, \sum, S, P)$, gdzie $P = P_1\cup P_2\cup P_3$ przy czym:

- $P_1 = \{S\rightarrow[q_0Z_0q]$ dla każdego $q\in Q\}$

  Gramatyka akceptuje takie słowa, które pozwalają na przejście ze stanu startowego q_0 do dowolnego stanu q przy usunięciu ze stosu symbolu początkowego Z_0 (w stanie q stos jest pusty). Gramatyka akceptuje więc słowa rozpoznawane przez A.
- $P_2 = \{[pZq] \rightarrow a\ [r_1Z_1r_2][r_2Z_2r_3]...[r_nZ_nq]\ \mid (r_1,Z_1Z_2...Z_n) \in\delta(p,a,Z)$, po wszystkich możliwych wariacjach stanów pośrednich $(r_1r_2...r_n) \mid r_i\in Q$ oraz dla każdego symbolu terminalnego $a\in\sum\cup\{\varepsilon\}$, dowolnych symboli na stosie $Z,Z_1,Z_2,...,Z_n\in\Gamma$ oraz dowolnych stanów krańcowych $p,q\in Q\}$.

  Ten zestaw produkcji pozwala gramatyce G na generację słów złożonych z symboli terminalnych postaci $aw_1w_2...w_n$ gdzie $(p,aw_1w_2...w_nx,Z\alpha)$ $\neg$ $(r_1,w_1w_2...w_nx,Z_1Z_2...Z_n\alpha)$ $\neg^*$ $(r_2,w_2...w_nx,Z_2...Z_n\alpha)$ $\neg^*$ ... $\neg^*$ $(r_n,w_nx,Z_n\alpha)$ $\neg^*(q,x,\alpha)$.
- $P_3 = \{[pZq] \rightarrow a \mid (q,\varepsilon)\in\delta(p,a,Z)$, dla każdego symbolu terminalnego $a\in\sum\cup\{\varepsilon\}$, dowolnych stanów krańcowych $p,q\in Q$, oraz dowolnego $Z\in\Gamma\}$.

  Ten zestaw produkcji pozwala gramatyce G na generację symboli terminalnych wtedy gdy automat takie symbole akceptuje.

Teraz wystarczy udowodnić, że każde słowo w jest generowane przez G wtedy i tylko wtedy, gdy w jest akceptowane przez A. Formalnie:

$S\Rightarrow^* w \quad\Leftrightarrow\quad (q_0,w,Z_0)\neg^*(q,\varepsilon,\varepsilon)$

Powyższa zależność jest oczywista (wynika z konstrukcji G: każde przejście A posiada równoważną produkcję gramatyczną oraz każda akceptacja symbolu automatu powoduje wygenerowanie symbolu przez gramatykę), dlatego nie będziemy rozpisywali jej dowodu.

■

4.4 Deterministyczny automat ze stosem

Deterministyczny automat ze stosem jak sama nazwa wskazuje ma z góry określone struktury, które akceptuje; nie istnieje w nim żadna możliwość

wyboru spośród dopuszczalnych ruchów. Nakłada to na funkcję przejścia δ pewne restrykcje:

- $\#\delta(q,a,Z) \leq 1$ dla każdego $q \in Q$, $a \in \sum \cup \{\varepsilon\}$, $Z \in \Gamma$. Innymi słowy, jeśli na stosie jest Z, a na wejściu automatu oczekuje a, to automat może przejść tylko do ściśle określonego stanu przy dokonaniu ściśle określonej operacji na stosie
- $\#\delta(q, \varepsilon, Z) > 0 \Rightarrow \#\delta(q,a,Z) = 0$ dla każdego $q \in Q$, $a \in \sum \cup \{\varepsilon\}$, $Z \in \Gamma$. Innymi słowy, jeśli automat w stanie q może wykonać operację na stosie bez akceptacji symbolu wejściowego, to nie może istnieć żadne przejście automatu z tego stanu, gdy na stosie jest ten sam symbol, a symbol wejściowy jest akceptowany.

Mimo, iż wydawać by się mogło, że skończone automaty ze stosem są rzadko stosowane z uwagi na swoje ograniczenie funkcji przejścia, w rzeczywistości jest zupełnie odwrotnie. Programy typu YACC generujące automaty rozpoznające języki programowania są oparte właśnie o deterministyczne automaty ze stosem. Wobec tego pytanie jakie języki są akceptowane przez deterministyczne automaty ze stosem sprowadza się do pytania jaka składnia języka programowania może zostać jednoznaczną kompilację.

Dla deterministycznego automatu ze stosem akceptacja poprzez pusty stos jest stosunkowo trudna; Jeśli taki automat akceptowałby słowo w, wówczas nie mógłby już zaakceptować żadnego słowa wx (słowo w wydłużone o ciąg symboli terminalnych). Dzieje się tak, gdyż akceptacja w przez pusty stos w dowolnym stanie jest równoważna opróżnieniu stosu w tym stanie. Tym samym (z definicji deterministycznego automatu skończonego) automat nie mógłby przejść do żadnego innego stanu pod wpływam symbolu wejściowego, więc skończyłby działanie. Wniosek: taki automat może zostać zastosowany tylko do rozpoznawania języków L posiadających własność prefiksową: jeśli $w \in L$, to $wx \notin L$ dla dowolnego $x \neq \varepsilon$.

Jakkolwiek, kompilatory dokonują akceptacji przez opróżnienie stosu. Trik polega na tym, że każde słowo rozpoznawanego języka zakończone jest specjalnym ogranicznikiem \$. Tym samym automat zaakceptuje słowo jeśli zaakceptuje też następujący po nim ogranicznik. W przypadku braku ogranicznika wiadomo, że słowo jeszcze się nie zakończyło. W ten sposób można przy pomocy deterministycznych automatów ze stosem rozpoznawać przez pusty stos języki nie posiadające własności prefiksowej.

Jeśli L jest językiem regularnym, to jest także językiem akceptowanym przez deterministyczny automat ze stosem; Automat taki po wyeliminowaniu stosu może z powodzeniem symulować dowolny automat skończony.

5. Analiza języków bez-kontekstowych

Streszczenie: *Na początku tego rozdziału przedstawione zostaną algorytmy czyszczące gramatyki bez-kontekstowe, czyli usuwające z nich symbole zbędne. W dalszej części pokazany zostanie schemat działania przy sprowadzaniu dowolnej gramatyki bez-kontekstowej do bardzo ważnej, postaci normalnej Chomsky'ego. Udowodniony zostanie także lemat o pompowaniu dla gramatyk bez-kontekstowych służący do rozstrzygania, czy dany język może być bez-kontekstowy. Na końcu rozdziału omówione zostaną własności zbioru języków bez-kontekstowych, w szczególności problem domknięcia na różne operacje algebry zbiorów , pustości oraz przynależności słowa do gramatyki. Do rozwiązania tego ostatniego problemu pokazany zostanie algorytm oparty na metodzie programowania dynamicznego.*

5.1 Czyszczenie gramatyk bez-kontekstowych

Gramatyki bez-kontekstowe mogą być w znacznym stopniu uproszczone poprzez ich oczyszczenie. To podstawowych czynności czyszczących należy:

1. Pozbycie się symboli *bezużytecznych*, czyli takich, które nie występują w wywodzie żadnego słowa złożonego z symboli terminalnych.
2. Pozbycie się *ε-produkcji*, czyli produkcji typu $v \rightarrow \varepsilon$, gdzie v jest nie-terminalem. Innymi słowy gramatyka traci możliwość generowania ε jako słów języka.
3. Pozbycie się produkcji *zbędnych* typu $v_1 \rightarrow v_2$, gdzie v_1 oraz v_2 są różnymi symbolami nie-terminalnymi.

Oprócz tego często gramatyki doprowadza się do *postaci normalnej Chomsky'ego*, czyli takiej, w której produkcje są postaci $v \rightarrow v_1 v_2$ oraz $v \rightarrow a$, gdzie v, v_1, v_2 są symbolami nie-terminalnymi natomiast a jest symbolem terminalnym.

Pozbycie się symboli bezużytecznych

Aby symbol X był użyteczny, musi on spełniać (w poniższej kolejności) dwa warunki:

1. X musi wywodzić słowo terminalne (lub być symbolem terminalnym). Formalnie: $X \Rightarrow^* a$, gdzie a jest symbolem terminalnym.
2. X musi być *osiągalne* z symbolu startowego S. Formalnie: $S \Rightarrow^* \alpha X \beta$. Uwaga: X nie będzie użyteczny, jeśli α lub β zawierają symbol nie

spełniający (1). Ważne jest zatem by najpierw pozbyć się produkcji, w których występują symbole nie spełniające (1) a następnie pozbyć się symboli nieosiągalnych.

Znajdowanie symboli, które nie wywodzą słowa terminalnego:

Niech V będzie zbiorem symboli nie-terminalnych. Docelowo będziemy chcieli, aby $U \subset V$ zawierał symbole użyteczne. W celu znalezienia wszystkich elementów U posługujemy się następującym schematem indukcyjnym:

- *Baza*: Symbol terminalny z pewnością wywodzi symbol terminalny, więc przyjmujemy $U = \{v \in V \mid (v \rightarrow a) \in P \text{ oraz } a \in \sum\}$
- *Indukcja*: Dla każdej produkcji postaci: $A \rightarrow X_1X_2...X_n$ jeśli $\forall_i\ X_i \in U$ to również $A \in U$. Innymi słowy, jeśli każdy symbol z ciała produkcji wywodzi symbol terminalny, to głowa tej produkcji też wywodzi symbol terminalny

Powyższy schemat jest powtarzany tak długo, jak da się znaleźć produkcję, która spowoduje powiększenie zbioru U.

Przykład: Rozpatrzmy gramatykę o następujących produkcjach:

$S \rightarrow AB|C|DED$; $A \rightarrow 0B|C$; $B \rightarrow 1|A0$; $C \rightarrow AC|C1$; $E \rightarrow 0$

Krok 1) Symbolami terminalnymi są 0 oraz 1

Krok 2) $B \rightarrow 1$ oraz $E \rightarrow 0$, więc $U=\{B,E\}$

Krok 3) $A \rightarrow 0B$, więc $U=\{B,E,A\}$

Krok 4) $S \rightarrow AB$, więc $U=\{B,E,A,S\}$

Krok 5) U nie może być powiększone, czyli zbiór symboli bezużytecznych $U_n=V-U=\{D,E,C\}$. Produkcje zawierające symbole bezużyteczne są eliminowane. Pozostają tylko $S \rightarrow AB$; $A \rightarrow 0B$; $B \rightarrow 1|A0$;

Znajdowanie symboli, osiągalnych ze stanu startowego S:

Niech $W=V \cup \sum$ będzie zbiorem symboli terminalnych i nie-terminalnych. Docelowo będziemy chcieli, aby $U \subset W$ zawierał symbole osiągalne od symbolu startowego S. W celu znalezienia wszystkich elementów U posługujemy się następującym schematem indukcyjnym:

- *Baza*: Symbol startowy musi być w U, czyli $U=\{S\}$
- *Indukcja*: Jeśli w U jest symbol A, to każdy symbol X_i z ciała produkcji $A \rightarrow X_1X_2...X_n$ także jest w U.

Powyższy schemat jest powtarzany tak długo, jak da się znaleźć nowe symbole osiągalne z symboli należących do U. W ostateczność otrzymamy zbiór symboli nieosiągalnych $U_n=W-U$. Teraz wystarczy usunąć z gramatyki te produkcje, których głowa jest nieosiągalna. Dodatkowo możemy usunąć ze

alfabetu terminalnego symbole należące do U_n, jako że nie zostaną one nigdy wygenerowane.

Przykład: Poddając analizie symboli nieosiągalnych gramatykę $S\rightarrow AB$; $A\rightarrow 0B$; $B\rightarrow 1|A0$ mamy:

Krok 1) Symbolem osiągalnym jest S; U={S}

Krok 2) $S\rightarrow AB$ więc U={S,A,B}

Krok 3) $A\rightarrow 0B$, $B\rightarrow 1$ więc U={S,A,B,0,1}

Krok 4) U nie może być powiększone, i skoro $U_n = \varnothing$ wszystkie symbole są osiągalne

Przykład niepoprawnego stosowania algorytmu eliminacji symboli bezużytecznych:

Rozpatrzmy jeszcze raz gramatykę posiadającą następujący zbiór produkcji: $S\rightarrow AB|C|DED$; $A\rightarrow 0B|C$; $B\rightarrow 1|A0$; $C\rightarrow AC|C1$; $E\rightarrow 0$. Teraz jednak oczyśćmy ją najpierw pod kątem warunku (2), a potem pod kątem warunku (1). Jeśli wyeliminujemy na początku symbole nieosiągalne z S, to zbiór produkcji pozostanie taki sam. Stosując teraz eliminację symboli, które nie wywodzą słowa terminalnego otrzymamy zbiór U_n={C,D} (E należy do U, ponieważ $E\rightarrow 0$). Ostatecznie zbiór produkcji ograniczy się do $S\rightarrow AB$; $A\rightarrow 0B$; $B\rightarrow 1|A0$; $E\rightarrow 0$ i wciąż będzie zawierał symbol bezużyteczny E.

Pozbycie się ε-produkcji

Symbol nie-terminalny v nazwiemy *zerującym*, jeśli można z niego za pomocą produkcji gramatyki wywieźć symbol pusty ε, czyli jeśli $v\Rightarrow^{*}\varepsilon$. Oznaczmy przez U zbiór symboli zerujących gramatyki. Poniższy schemat indukcyjny znajduje wszystkie elementy U:

- *Baza*: $U = \{v \mid (v\rightarrow\varepsilon)\in P\}$
- *Indukcja*: Jeśli A jest głową produkcji, której ciało składa się z symboli zerujących, wówczas A też jest symbolem zerującym. Formalnie podstaw za U zbiór
  $U \cup \{A \mid (A\rightarrow X_1X_2...X_n)\in P$, przy czym $\forall_{i\in\{1,2,...,n\}}\ X_i \in U\}$

Powyższy schemat jest powtarzany tak długo, jak da się znaleźć nowe symbole zerujące. Teraz wystarczy wykonać dwie czynności by pozbyć się ε-produkcji:

1. Dla każdej produkcji z P postaci $A\rightarrow X_1X_2...X_k$ dodaj do P wszystkie produkcje, które mogą być utworzone poprzez wyeliminowanie kilku lub wszystkich X_i, które są zerujące. Równocześnie jeśli wszystkie $X_1X_2...X_k$ są zerujące nie dodawaj produkcji powstałej z wyeliminowania wszystkich zmiennych. Przykładowo jeśli $A\rightarrow BC$ i B oraz C są symbolami zerującymi, dodaj produkcje $A\rightarrow B$ oraz $A\rightarrow C$. Formalnie dla

produkcji $A \rightarrow X_1X_2...X_k$, niech $M = \{m \mid A_m$ jest symbolem zerującym$\}$. Dla tej produkcji do zbioru P produkcji gramatyki należy dołączyć następujący zbiór produkcji: $P' = \{A \rightarrow z_1z_2...z_k \mid z_i = X_i$ gdy $i \notin M$; $z_i \in \{X_i, ""\}$ gdy $i \in M\} - \{A \rightarrow \varepsilon\}$

2. Usunąć ze zbioru P wszelkie produkcje wymazujące, których głową nie jest początek gramatyki, czyli należące do zbioru $\{v \rightarrow \varepsilon \mid v \in V - \{S\}\}$.
3. Jeśli symbolem zerującym być S, dodaj do P produkcję $\{S \rightarrow \varepsilon\}$

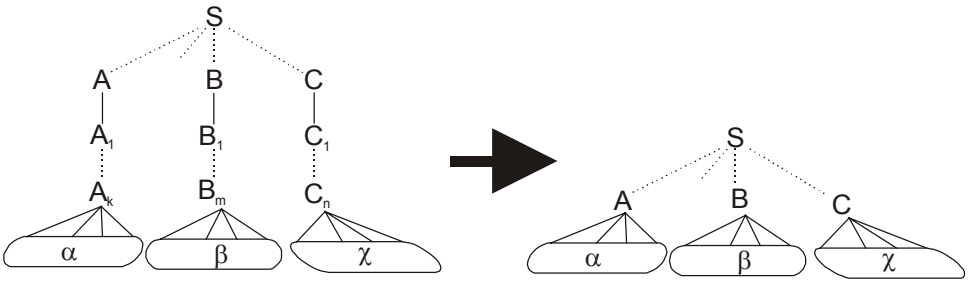
Łatwo udowodnić, że tak zmodyfikowana gramatyka, która nie posiada już produkcji wymazującej dalej generuje ten sam język. Wywód dowolnego słowa w pierwotnej gramatyce można schematycznie zaznaczyć jako: $S \Rightarrow^* \alpha A \gamma \Rightarrow \alpha B \beta \gamma \Rightarrow^* \alpha \beta \gamma \Rightarrow^*$ w, jeśli $A \rightarrow \varepsilon$ jest produkcją wymazującą, a $A \rightarrow B\beta$ zwyczajną (nie wymazującą) produkcją. Jako że B jest symbolem zerującym w zmienionej gramatyce zamiast tych dwóch produkcji będzie produkcja $A \rightarrow \beta$ co pozwoli nam na wywód słowa w: $S \Rightarrow^* \alpha A \gamma \Rightarrow \alpha \beta \gamma \Rightarrow^*$ w.

Warto zauważyć, że po dokonaniu ε-eliminacji w gramatyce mogą pojawić się symbole bezużyteczne: jeśli zarówno A jak i B są symbolami zerującymi, a produkcje gramatyki $S \rightarrow AB$; $A \rightarrow \varepsilon$; $B \rightarrow \varepsilon$; zostały zamienione na produkcje $S \rightarrow A|B|\varepsilon$ to zarówno A jak i B staną się symbolami bezużytecznymi.

Pozbycie się produkcji zbędnych

Pozostaje nam omówić oczyszczanie gramatyki polegające na pozbyciu się produkcji zbędnych. Realizowana jest ona w następujący sposób:

1. Eliminacja ε-produkcji
2. Eliminacja symboli bezużytecznych
3. Wykrycie wszystkich par zmiennych (A,B) takich, że $A \Rightarrow^* B$. Z uwagi na to, że nie ma już ε-produkcji, takie pary (A,B) określają symbole zbędne na drodze wywodu. Produkcje zbędne są znajdowane przez przeglądanie drzewa wywodu począwszy od głowy gramatyki.
4. Jeśli w drzewie wywodu istnieje ścieżka od węzła A w głąb drzewa bez rozgałęzień to mamy do czynienia z produkcjami zbędnymi. Niech ścieżka bez rozgałęzień przechodzi przez węzły A, A_1, A, ..., A_k a następnie rozgałęzia się na wierzchołki tworzące w sumie ciąg α. Wywód ten może być zapisany jako: $A \Rightarrow A_1 \Rightarrow A_2 \Rightarrow ... \Rightarrow A_k \Rightarrow \alpha$. Wywód korzysta z podzbioru produkcji gramatyki $\{A \rightarrow A_1; A_1 \rightarrow A_2; ... ; A_{k-1} \rightarrow A_k; A_k \rightarrow \alpha\}$. Każdy podzbiór produkcji tej postaci musi więc zostać zamieniony na zbiór $\{A \rightarrow \alpha\}$. Graficzna ilustracja tej zamiany pokazana jest na rysunku nr 13.



Rysunek 13 Eliminacja produkcji zbędnych

Postać normalna Chomsky'ego

W celu doprowadzenia gramatyki do postaci normalnej Chomsky'ego wyjdziemy od dowolnej gramatyki G, która została oczyszczona tzn. usunięte z niej zostały ε-produkcje, symbole bezużyteczne oraz produkcje zbędne. Niech więc G=(V,∑,P,S). Zbudujemy najpierw tzw. kopię alfabetu, tzn. zbiór symboli nie-terminalnych, w którym każdy nie-terminal związany jest z innym terminalem ze zbioru ∑. Formalnie:

- $V_\Sigma = \{V_a \mid V_a$ jest nie-terminalem oraz $a \in \Sigma\}$

Niech wyjściową gramatyką będzie $H=((V \cup V_\Sigma),\Sigma,P_H,S)$ Zbiór produkcji PH jest następujący: zaczynamy od dodania do niego wszystkich produkcji z P, przy czym w każdej produkcji zamieniamy każdy terminal na odpowiadający mu nie-terminal z kopii alfabetu. W uproszczeniu: $P_H = \{v \rightarrow v_a v_1 v_b v_2 v_c ... \mid (v \rightarrow a v_1 b v_2 c ...) \in P\}$. Teraz dodajemy do PH produkcje pozwalające na zamianę nie-terminalów z kopii alfabetu na odpowiadające im terminale: w miejsce P_H podstawiamy $P_H \cup \{v_a \rightarrow a \mid v_a \in V_\Sigma$ jest związany z $a \in \Sigma\}$.

Teraz pozostaje nam już tylko porozcinać produkcje w P_H by ich ciała zawierały albo 1 terminal albo 2 nie-terminale. Prawa, których ciałem jest terminal są zgodne z prawami w postaci normalnej Chomsky'ego. Musimy więc rozciąć prawa postaci $v \rightarrow v_1 v_2 ... v_n$. Rozcięcie dla prawa, którego ciało posiada n nie-terminali polega na dodaniu do zbioru nie-terminali (n-2) nowych nie-terminali $u_1,u_2,...,u_{n-2}$ i zamianie produkcji $v \rightarrow v_1 v_2 ... v_n$ na zestaw produkcji: $\{v \rightarrow v_1 u_1;\ u_1 \rightarrow v_2 u_2;\ ...;\ u_{n-2} \rightarrow v_{n-1} v_n\}$. W wyniku tak przeprowadzonych rozcięć produkcje są postaci $v \rightarrow v_1 v_2$ oraz $v \rightarrow a$, gdzie v,v_1,v_2 są symbolami nie-terminalnymi natomiast a jest symbolem terminalnym.

5.2 Lemat o pompowaniu dla gramatyk bez-kontekstowych

Lemat Chomsky'ego o pompowaniu dla gramatyk bez-kontekstowych brzmi następująco:

Jeśli L_1 jest językiem bez-kontekstowym, wówczas istnieje taka stała N, że każde słowo z L_1 o długości większej od N może być zapisane jako $w=u_1w_1uw_2u_2$, gdzie:

1. $| w_1uw_2| \leq N$
2. $w_1w_2 \neq \varepsilon$ (co najmniej jedno z w_1, w_2 nie może być puste)
3. $u_1w_1^i uw_2^i u_2 \in L_1$ dla $i = 0,1, ...$

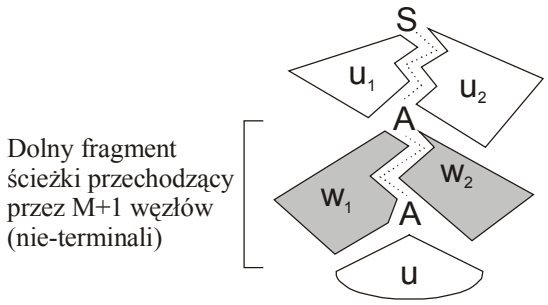
Dowód:

□

L_1 jest językiem bez-kontekstowym to może zostać wygenerowany przez gramatykę Chomsky'ego. Niech gramatyka ta zawiera M symboli nie-terminalnych. Przyjmijmy $N=2^M$.

- W gramatyce Chomsky'ego ciała produkcji mają co najwyżej 2 nie-terminale, więc łatwo spostrzec, że dla każdego słowa w długości ≥ N wygenerowanego przez tą gramatykę, w jego drzewie wywodu będzie istniała ścieżka przechodząca przez co najmniej M+1 nie-terminali. Dowód indukcyjny ze względu na długość słowa w:
  - *Baza*: słowo „a" długości 1 jest generowane przez 1 produkcję, więc istnieje w jego drzewie wywodu ścieżka długości $\log_2(1) + 1 = M+1$.
  - *Indukcja*: słowo w długości n+1 ma wywód $S \rightarrow v_1v_2 \Rightarrow^* \alpha v_2 \Rightarrow^* \alpha\beta = w$. Stąd (założenie indukcyjne) słowo α długości k<n ma ścieżkę wywodu z v_1 długości co najmniej $\log_2(k) + 1$. Podobnie słowo β długości n-k<n ma ścieżkę wywodu z v_2 długości co najmniej $\log_2(n-k) + 1$. Zatem w drzewie wywodu w istnieje ścieżka wywodu długości co najmniej $\log_2(\lceil n/2 \rceil) +1+1 \leq \log_2(n) + 1$ c.b.d.u.

Zatem słowo o długości ≥ N posiada ścieżkę długości co najmniej M+1. Skoro M jest liczbą nie-terminali, zatem na tej ścieżce jakiś nie-terminal musi się powtórzyć. Rozpatrzmy drzewo wywodu słowa w pokazane na rysunku nr 14 i jego najdłuższą ścieżkę wywodu długości co najmniej M+1. Skupmy się na dolnej części tej ścieżki (tzn. na najniższych M+1 węzłach). Wśród tych węzłów jakiś węzeł (np. nie-terminal A) powtórzy się. Niech dolne A generuje słowo u oraz niech górne A generuje słowo w_1uw_2. Skoro począwszy od górnego A ścieżka ma długość co-najwyżej M+1, zatem (z postaci normalnej Chomsky'ego) słowo w_1uw_2 ma długość co-najwyżej $2^{M+1-1} = 2^M = N$ (dowód 1 własności lematu).



Rysunek 14 Lemat o pompowaniu dla gramatyk bez-kontekstowych

Ponadto jeśli górne A nie generuje terminala, to musi (z postaci normalnej Chomsky'ego) generować 2 nie-terminale, z których jeden nie leży na ścieżce łączącej górne i dolne A. Stąd $w_1w_2 \neq \varepsilon$ (dowód 2 własności lematu).

Jeśli w miejsce górnego A generującego w_1uw_2 wstawimy dolne A generujące u, to od tej pory górne A będzie generowało tylko u. Czyli jeśli pierwotnie $w = u_1w_1uw_2u_2$ to po tej zamianie otrzymamy drzewo wywodu słowa u_1uu_2. Wniosek: $u_1w_1^0uw_2^0u_2 \in L_1$.

Jeśli w miejsce dolnego A generującego u wstawimy górne A generujące w_1uw_2, to od tej pory dolne A będzie generowało w_1uw_2. Czyli jeśli pierwotnie $w = u_1w_1uw_2u_2$ to po tej zamianie otrzymamy drzewo wywodu słowa $u_1w_1w_1uw_2w_2u_2 = u_1w_1^2uw_2^2u_2 \in L_1$. Powtarzając tą operację dla nowo powstałego dolnego A otrzymujemy słowa postaci $u_1w_1^2uw_2^2u_2 \in L_1$. Ostatecznie więc $u_1w_1^iuw_2^iu_2 \in L_1$ dla i=0,1,2... (dowód 3 własności lematu). ■

Przykład zastosowania lematu o pompowaniu dla gramatyk bez-kontekstowych

Udowodnimy, że język $L=\{a^nb^nc^n \mid n\in\aleph\}$ nie jest bez-kontekstowy. W tym celu zakładamy nie-wprost, że L jest bez-kontekstowy. Dla dowolnego N, wybrane słowo długości $\geq$ N będzie miało postać: $w=a^Nb^Nc^N$. Rozpatrzmy podział słowa w na części $u_1w_1uw_2u_1$. Jeśli w_1 zachodzi na terminale a,b,c, to w_1^i zburzy porządek $a^nb^nc^n$. Jeśli w_1 zachodzi tylko na terminale a,b (w_1 = aa...abb...b), wtedy w_1^i zburzy porządek a^nb^n. Jeśli w_1 zachodzi tylko na terminal a, to aby $u_1w_1^iuw_2^iu_2$ posiadało tyle samo a,b oraz c, słowo w_2 musi zachodzić na b,c (w_2 = bb...bcc...c), wtedy jednak w_2^i zburzy porządek b^nc^n. Wniosek: dla jakiegokolwiek podziału $w=a^Nb^Nc^N$ na części $u_1w_1uw_2u_1$ słowo $u_1w_1^iuw_2^iu_2 \notin L$. Sprzeczność, czyli L nie jest bez-kontekstowy.

5.3 Własności rodziny języków bez-kontekstowych

Podstawienie

Definicja podstawienia s została dokładnie omówiona przy językach regularnych. Teraz udowodnimy twierdzenie o podstawieniu dla gramatyk bez-kontekstowych.

Jeśli podstawienie s przypisuje gramatykę bez-kontekstową każdemu symbolowi z alfabetu pewnej gramatyki bez-kontekstowej L, to s(L) jest także gramatyką bez-kontekstową.

Dowód:

 Niech gramatyką języka L będzie G=(V,∑,P,S). Dla każdego $a\in\sum$ oznaczmy przez L_a język bez-kontekstowy podstawiany w miejsce terminala a. Niech każdy taki język La będzie generowany przez odpowiadającą mu gramatykę bez-kontekstową $G_a=(V_a,\sum_a,P_a,S_a)$. Poprzez zamianę nazw symboli nie-terminalnych w każdej gramatyce G_a (włączając stany startowe gramatyk G_a) Możemy doprowadzić, by gramatyka G oraz G_a nie miały nie-terminali o tej samej nazwie.

Podstawienie s(L) oznacza, że w miejsce każdego słowa $w=a_1a_2...a_n$ wygenerowanego przez gramatykę G podstawiamy $s(w) = s(a_1)s(a_2)...s(a_n) =S_{a1}S_{a2}...S_{an}$. W miejsce każdego terminala a gramatyki G należy więc podstawić symbol startowy S_a gramatyki G_a. Taka konstrukcja pozwoli nam niewątpliwie wygenerować zbiór $\{s(w)\ \}\ w\in L\} = s(L)$. Pozostaje sprawdzić, czy taka konstrukcja tworzy gramatykę bez-kontekstową. Weźmy wywód dowolnego słowa $w\in L$. Niech D_w będzie drzewem tego wywodu. Podstawienie za każdy terminal $a\in\sum$ symbolu S_a spowoduje, że do liści D_w zostaną dołączone pod-drzewa z korzeniami S_a. Każde takie pod-drzewo będzie kończyło się liśćmi ze zbioru $\sum_a$ jako że L_a są także bez-kontekstowe. Cała konstrukcja będzie zatem dalej drzewem. Stąd wniosek, że dla każdego słowa $w=a_1a_2...a_n$, słowo s(w) będzie miało wywód: $S\Rightarrow^*S_{a1}S_{a2}...S_{an}\Rightarrow^* s(a_1)s(a_2)...s(a_n) = s(w)$. Skoro więc każde słowo s(w) języka s(L) ma drzewo wywodu, więc i sam język s(L) jest bez-kontekstowy. ■

Przykład

Rozpatrzmy język bez-kontekstowy $L=\{0^n1^n \mid n \geq 1\}$ generowany przez gramatykę G o produkcjach $S\rightarrow 0S1 \mid 01$. Za każdy symbol terminalny tego języka podstawimy inny język bez-kontekstowy, czyli np.:

- $s(0) = \{a^nb^m \mid m \leq n\}$ generowany przez gramatykę G_0: $S\rightarrow aSb \mid A$; $A\rightarrow aA \mid ab$
- $s(1) = \{ab,abc\}$ generowany przez gramatykę G_1: $S\rightarrow abA$; $A\rightarrow c \mid \varepsilon$

W celu usunięcia kolizji oznaczeń dla symbolów nie-terminalnych dla s(0) oraz s(1) zamieniamy w G_0: S na S_0; A na A_0 oraz w G_1: S na S_1. Gramatyki mają teraz postać:

G: $S \rightarrow 0S1 \mid 01$

G_0: $S_0 \rightarrow aS_0b \mid A_0$; $A_0 \rightarrow aA_0 \mid ab$

G_1: $S_1 \rightarrow abA$; $A \rightarrow c \mid \varepsilon$

Zamieniając w gramatyce G symbol 0 na S_0 oraz 1 na S_1 otrzymujemy ostatecznie gramatykę:

s(L): $S \rightarrow S_0SS_1 \mid 01$

$S_0 \rightarrow aS_0b \mid A_0$; $A_0 \rightarrow aA_0 \mid ab$

$S_1 \rightarrow abA$; $A \rightarrow c \mid \varepsilon$

która generuje język $s(L) = \{ (a^ib^j)^n (ab+abc)^n \quad j \leq i; n \geq 1\}$

Konsekwencja domknięcia przez podstawienie

Fakt, że podstawienie za każdy symbol alfabetu języka bez-kontekstowego innego języka bez-kontekstowego jest dalej językiem bez-kontekstowym pozwala nam dowieść, że rodzina języków bez-kontekstowych jest domknięta ze względu na sumę, konkatenację, powielanie (operator *) oraz homomorfizm.

- Suma: niech L={a, b} jest bez-kontekstowy oraz bez-kontekstowe są L_1 oraz L_2. Jeśli $s(a) = L_1$ oraz $s(b) = L_2$, to $s(L)=L_1 \cup L_2$ jest bez-kontekstowy.
- Konkatenacja: niech L={ab} jest bez-kontekstowy oraz bez-kontekstowe są L_1 oraz L_2. Jeśli $s(a) = L_1$ oraz $s(b) = L_2$, to $s(L)=L_1L_2$ jest bez-kontekstowy.
- Powielanie (*): niech $L=\{a^*\}$ oraz L_1 są bez-kontekstowe. Jeśli $s(a) = L_1$, to $s(L)=L_1^*$ jest bez-kontekstowy.
- Homomorfizm. Niech L=L(G), gdzie $G=(V,\sum,P,S)$ jest bez-kontekstowa. Niech funkcja homomorfizmu będzie zdefiniowana jako $h: \sum \rightarrow \sum_h$. Dla każdego symbolu terminalnego $a \in \sum$ definiujemy język bez-kontekstowy $L_a = \{h(a)\}$. Stosując podstawienie s(L) otrzymujemy h(L) co na mocy twierdzenia o podstawieniu jest językiem bez-kontekstowym.

Języki bez-kontekstowe nie są domknięte ze względu na przecięcie

Dla udowodnienia tej zależności wystarczy podać przykład dwóch języków bez-kontekstowych, których przecięcie nie jest językiem bez-kontekstowym. Przyjmijmy za $L_1=\{a^nb^nc^k; n, k \in \aleph\}$ generowany przez: $S \rightarrow S_1C$; $S_1 \rightarrow aS_1b|ab$; $C \rightarrow cC|c$ oraz
$L_1=\{a^kb^nc^n; n, k \in \aleph\}$ generowany przez: $S \rightarrow AS_1$; $S_1 \rightarrow bS_1c|bc$; $A \rightarrow aA|a$.

Do przecięcia $L_1 \cap L_2$ należą te słowa z L_1 lub L_2, które spełniają dwie własności: ilość a jest równa ilości b oraz ilość b jest równa ilości c. Stąd $L_1 \cap L_2 = \{a^n b^n c^n;\ n, k \in \aleph\}$ – co nie jest językiem bez-kontekstowym (patrz: lemat o pompowaniu).

Niektóre, prostsze języki bez-kontekstowe mogą jednak w przecięciu dać język bez-kontekstowy, np. $L_1 = \{a^n b^n c^k;\ n, k \in \aleph\}$ oraz $L_1 = \{a^n b^m c^k;\ n, k, m \in \aleph;\ m \le n\}$. $L_1 \cap L_2 = L_1$

Języki bez-kontekstowe nie są domknięte ze względu na dopełnienie

Języki bez-kontekstowe są domknięte ze względu na sumę, więc jeśli byłyby domknięte ze względu na dopełnienie, to (prawa DeMorgana) byłyby także domknięte ze względu na przecięcie, sprzeczność. Dla przykładu weźmy języki:

- $L_1 = \{a^n b^i c^j;\ n, i, j \in \aleph;\ i<n;\}$ generowany przez $S \to S_1C$; $S_1 \to aS_1b|S_2$; $S_2 \to S_2a|a$; $C \to Cc|c$ oraz
- $L_2 = \{a^n b^i c^j;\ n, i, j \in \aleph;\ i>n;\}$ generowany przez $S \to S_1C$; $S_1 \to aS_1b|S_2$; $S_2 \to S_2b|b$; $C \to Cc|c$.
- Suma tych języków (udowodnione) jest językiem bez-kontekstowym. Czyli $L_{ab} = L_1L_2 = \{a^n b^i c^j \mid n, i, j \in \aleph;\ i \ne n\}$ jest językiem bez-kontekstowym.
- Podobnie możemy dowieść, że $L_{ac} = \{a^n b^i c^j \mid n,i,j \in \aleph;\ j \ne n\}$ i $L_{bc} = \{a^n b^i c^j \mid n,i,j \in \aleph;\ i \ne j\}$ są bez-kontekstowe. Suma $L_{ac} \cup L_{ab} \cup L_{bc} = \{a^n b^i c^j \mid n, i, j \in \aleph;\ i \ne j;\ i \ne n;\ n \ne j\}$ jest więc językiem bez-kontekstowym. Dopełnienie $(L_{ac} \cup L_{ab} \cup L_{bc})^{-1} = \{a^n b^n c^n \mid n \in \aleph\}$ nie jest natomiast językiem bez-kontekstowym.

Języki bez-kontekstowe są domknięte ze względu na odwrócenie

Dla dowolnego słowa $a_1a_2...a_n = w \in \sum^*$, odwrócenie słowa $\overleftarrow{w} = a_n a_{n-1}...a_1$. Język odwrócony do L definiujemy następująco: $\overleftarrow{L} = \{\overleftarrow{w} \mid w \in L\}$. Jeśli L jest bez-kontekstowy, to $\overleftarrow{L}$ także jest bez-kontekstowy. By z L=L(G) otrzymać $\overleftarrow{L} = L(\overleftarrow{G})$ wystarczy każdą produkcję gramatyki G postaci $v \to \alpha$, gdzie $v \in V$; $\alpha \in (\sum \cup V)^*$ zamienić na $v \to \overleftarrow{\alpha}$. Każda tak zamieniona produkcja może należeć do gramatyki bez-kontekstowej $\overleftarrow{G} = (V, \sum, \overleftarrow{P}, S)$.

Sprawdzanie pustości języka bez-kontekstowego

W przypadku języków regularnych sprawdzanie, czy język jest pusty sprowadzało się do pytania, czy może on być reprezentowany przez wyrażenie

regularne ∅. Dla języków bez-kontekstowych sprawdzanie czy są one puste może być dokonywane przy pomocy jednej z reprezentacji języka: gramatyki bez-kontekstowej lub automatu ze stosem. Z uwagi na to, że są algorytmy pozwalające na przejście z jednej reprezentacji do drugiej, wybór reprezentacji jest obojętny. W przypadku reprezentacji przez gramatykę bez-kontekstową wystarczy sprawdzić, czy symbol startowy S gramatyki jest bezużyteczny. Jeśli tak, gramatyka generuje język pusty.

Sprawdzanie skończoności języka bez-kontekstowego

Niech L będzie dowolnym językiem bez-kontekstowym.

- Istnieje zatem pewna stała N z lematu o pompowaniu dla gramatyk bez-kontekstowych.
- Sprawdzaj wszystkie słowa długości pomiędzy N a 2N-1 czy należą do L (algorytm sprawdzania przynależności słowa podany jest poniżej).
- Jeśli jest słowo o długości pomiędzy N a 2N-1 należące do L to może na mocy lematu o pompowaniu być w nieskończoność pompowane generujące nieskończoną ilość słów. Wówczas język L jest nieskończony.
- Jeśli nie ma słowa o długości pomiędzy N a 2N-1 należącego do L, to N-1 jest górną granicą na długość słowa w tym języku (L jest więc skończony) gdyż:
    - Jeśli istnieje słowo $w=u_1w_1uw_2u_1$ długości co najmniej 2N należące do L, można wyodrębnić z tego słowa na mocy lematu o pompowaniu pod-słowo u_1uu_2 należące do L (tak wyodrębnione pod-słowo jest krótsze od w co najwyżej o N). Jeśli więc jest słowo dłuższe od 2N należące do L, można powtarzając schemat: [wytnij ze słowa w pod-słowa w_1 oraz w_2 i podstaw zredukowane słowo za w] znaleźć ostatecznie słowo o długości pomiędzy N a 2N-1 należące do L. W konsekwencji jeśli istnieje w L słowo o długości większej od N, język L jest nieskończony.

Sprawdzanie przynależności słowa do języka bez-kontekstowego

Symulowanie automatu ze stosem dla ciągu wejściowego w nie jest dobrym sposobem na sprawdzenie przynależności słowa do języka bez-kontekstowego akceptowanego przez ten stos. Dzieje się tak, gdyż automat ze stosem może być tak zaprojektowany, że w nieskończoność powiększa swój stos przy aktualnie badanym symbolu ε. W rezultacie badanie akceptowalności słowa w może się nigdy nie zakończyć mimo iż automat może być deterministyczny.

Istnieje algorytm zaproponowany przez Cocke-Younger-Kasami (CYK), który używając techniki programowania dynamicznego sprawdza w czasie $O(n^3)$ czy słowo $w=a_1a_2...a_n$ długości n należy do języka generowanego przez

gramatykę bez-kontekstową (w postaci normalnej Chomsky'ego). Dla dowolnego słowa w algorytm działa następująco:

1. Buduje pustą tablicę dwuwymiarową T wielkości n × n. Numery kolumn tej tablicy będą związane z początkowymi pozycjami podciągów w słowie w. Numery wierszy będą związane z długościami poszczególnych podciągów. Przykładowo komórka $T_{wiersz,kolumna} = T_{3,7}$ będzie związana z podciągiem długości 3 rozpoczynającym się na 7 pozycji słowa w. W komórce $T_{i,j}$ będą się znajdować (jeśli takie są) symbole gramatyki, z których da się wywieść podciąg długości 3 rozpoczynający się na 7 pozycji słowa w. Każdą komórkę będziemy oznaczali jako X_{ab}, co oznaczać będzie podciąg słowa w rozciągający się od pozycji a do pozycji b. Pierwszy wiersz tablicy będzie więc miał komórki oznaczone jako: $X_{1,1}$, $X_{2,2}$, ..., $X_{n,n}$, drugi: $X_{1,2}$, $X_{2,3}$,...$X_{n-1,n}$, a n-ty komórkę $X_{1,n}$.
2. Wypełniamy tablicę wartościami:
   - *Baza*: wiersz 1 zawiera komórki dla pod-ciągów w długości 1, czyli w komórce $X_{i,i}$ będą się znajdowały takie symbole A, dla których istnieje produkcja $A \rightarrow a$, przy czym symbol a jest jednoelementowym podciągiem w począwszy od pozycji i do pozycji i.
   - *Indukcja*: Załóżmy, że wiemy już z czego można wywieść k-1 elementowe podciągi w. Chodzi nam teraz o wyznaczenie zbioru symbolów, z których można wywieść podciągi k elementowe. Rozpatrzymy dowolny podciąg k elementowy począwszy od i do j pozycji w słowie w (j=i+k-1). Podciąg $a_i a_{i+1} ... a_j$ może być wywiedziony z symbolu A, jeśli istnieje produkcja $A \rightarrow BC$, przy czym z B można wywieść dowolny prefiks $a_i a_{i+1} ... a_j$, natomiast z C można wywieść resztę $a_i a_{i+1} ... a_j$. Innymi słowy, A jest w $X_{i,j}$ jeśli:
     a. Istnieje wartość m z przedziału $i \leq m \leq j$
     b. W komórce Xi,m jest symbol B
     c. W komórce Xm+1,j jest symbol C
     d. Istnieje produkcja $A \rightarrow BC$

Pomocniczo przy każdym symbolu A w komórce $X_{i,j}$ możemy umieszczać numery dwóch komórek, których symbole stoją w ciele produkcji A. Formalnie jeśli np. w komórce $X_{1,5}$ jest $A:[X_{1,3}X_{4,5}]$ oznacza to, że aby wywieść podciąg $a_1a_2a_3a_4a_5$ można zastosować produkcję $A \rightarrow BC$, przy czym $B \in X_{1,3}$ oraz $C \in X_{4,5}$. A:[a] oznacza zamianę A na terminal a w wyniku produkcji $A \rightarrow a$.

Przykład

Rozpatrzmy, czy słowo w=aabb należy do gramatyki $S \rightarrow AS|SB|AB$; $A \rightarrow a$; $B \rightarrow b$. Tablica T dla tego słowa ma postać:

	A	**A**	**b**	**b**
Podciąg dł. 1	$(X_{1,1})$ A:[a]	$(X_{2,2})$ A:[a]	$(X_{3,3})$ B:[b]	$(X_{4,4})$ B:[b]
Podciąg dł. 2	$(X_{1,2})$	$(X_{2,3})$ S:$[X_{2,2}X_{3,3}]$	$(X_{3,4})$	
Podciąg dł. 3	$(X_{1,3})$ S:$[X_{1,1}X_{2,3}]$	$(X_{2,4})$ S:$[X_{2,3}X_{4,4}]$		
Podciąg dł. 4	$(X_{1,4})$ S:$[X_{1,1}X_{2,4}]$ S:$[X_{1,3}X_{4,4}]$			

Jak widać podciąg 4-elementowy, czyli całe słowo w=aabb posiada dwa różne wywody w zadanej gramatyce:

$S\rightarrow[X_{1,1}X_{2,4}]\rightarrow a[X_{2,3}X_{4,4}]\rightarrow a[X_{2,2}X_{3,3}]b\rightarrow aabb$ oraz

$S\rightarrow[X_{1,3}X_{4,4}]\rightarrow[X_{1,1}X_{2,3}]b\rightarrow a[X_{2,2}X_{3,3}]b\rightarrow aabb$

Skoro w komórkach $(X_{1,2})$ i $(X_{3,4})$ nie ma żadnych symboli, zatem ani pod-słowo aa, ani bb nie posiada wywodu w tej gramatyce.

6. Języki kontekstowe i Maszyna Turinga

Streszczenie: *Na początku tego rozdziału przedstawiony zostanie podział gramatyk Chomsky'ego oraz formalna definicja maszyny Turinga akceptującej język generowany przez dowolną gramatykę. Po wyjaśnieniu pojęcia stanu maszyny Turinga i udowodnieniu równoważności akceptacji maszyny Turinga przez stan końcowy oraz zatrzymanie przedstawione zostaną różne rodzaje maszyn Turinga. Poprzez pokazanie różnych technik konstruowania maszyny Turinga pokazane zostanie jak może ona skutecznie symulować maszynę zliczającą, licznik oraz komputer.*

6.1 Podział gramatyk Chomsky'ego

Chomsky zaproponował następujący podział gramatyk $G=(V,\Sigma,P,S)$:

1. **Gramatyki regularne tworzące języki L_3**. Każda produkcja takiej gramatyki jest postaci $v\rightarrow au$ lub $v\rightarrow ua$, gdzie $a\in\Sigma\cup\{\varepsilon\}$; $v\in V$; $u\in V\cup\{\varepsilon\}$.
2. **Gramatyki bez-kontekstowe tworzące języki L_2**. Każda produkcja takiej gramatyki jest postaci $v\rightarrow\alpha$, gdzie $v\in V$; $\alpha\in(V\cup\Sigma\cup\{\varepsilon\})^*$.
3. **Gramatyki kontekstowe tworzące języki L_1**. Każda produkcja takiej gramatyki jest postaci $\alpha v\beta\rightarrow\gamma$, gdzie $v\in V$; $\alpha,\beta\in(V\cup\Sigma\cup\{\varepsilon\})^*$; $\gamma\in(V\cup\Sigma)^+$ lub postaci $S\rightarrow\varepsilon$, gdzie S jest głową gramatyki. Zbiór produkcji gramatyki kontekstowej zawiera więc wszystkie możliwe produkcje oprócz produkcji wymazujących. Jedyną dozwoloną produkcją wymazującą jest $S\rightarrow\varepsilon$, czyli produkcja pozwalająca na wygenerowanie słowa pustego.
4. **Pozostałe gramatyki tworzące języki L_0**. Na produkcje tej gramatyki nie są nałożone żadne restrykcje.

Dla następujących zbiorów języków generowanych przez różnego typu gramatyki zachodzi:

$L_3 \subset L_2 \subset L_1 \subset L_0$.

Dowód $L_3 \subset L_2$ oraz $L_3 \neq L_2$ Pokazaliśmy już, że automat ze stosem w ogóle nie wykorzystujący stosu może z powodzeniem symulować dowolny automat skończony. Stąd wniosek, że każdy język regularny jest także bez-kontekstowy. Z kolei język $\{a^nb^n \mid n\in\aleph;\ a,b\in\Sigma\}$ jest bez-kontekstowy (generowany przez

gramatykę $S \rightarrow aSb|ab$) natomiast nie jest regularny (patrz lemat o pompowaniu dla języków regularnych).

Dowód $L_2 \subset L_1$ oraz $L_2 \neq L_1$ Pokazaliśmy już, że dowolna gramatyka bez-kontekstowa może być sprowadzona do postaci normalnej Chomsky'ego, w której to postaci nie ma produkcji wymazujących oprócz produkcji $S \rightarrow \varepsilon$. Stąd wniosek, że każda gramatyka bez-kontekstowa może zostać sprowadzona do równoważnej gramatyki kontekstowej. Z kolei język $\{a^n b^n c^n \mid n \in \aleph; a,b,c \in \sum\}$ jest kontekstowy (generowany przez gramatykę $S \rightarrow abc|aSBc$; $cB \rightarrow Bc$; $bB \rightarrow bb$) natomiast nie jest bez-kontekstowy (patrz lemat o pompowaniu dla gramatyk bez-kontekstowych).

Dowód $L_2 \subset L_1$ oraz $L_2 \neq L_1$. Każda gramatyka generująca język kontekstowy po dodaniu produkcji wymazującej będzie generowała język typu L_0. Z kolei można podać języki akceptowane przez Maszynę Turinga (rozpoznającą L_0) natomiast nierozpoznawalne przez automat liniowo ograniczony (rozpoznający L_1).

6.2 Definicja Maszyny Turinga (MT)

Maszyna Turinga podobnie jak automat liniowo ograniczony posiada skończony zbiór stanów kontrolnych. Ponadto MT posiada *taśmę*, na której może zapisywać/odczytywać, pełniącą zarazem funkcję urządzenia przechowującego nieograniczoną ilość danych. Taśma podzielona jest na *komórki*. Każda komórka przechowuje pojedynczy symbol należący do *alfabetu taśmy*. Taśma jest pół-skończona tzn. kończy się tylko z lewej strony. *Głowica* maszyny wskazuje na aktualną komórkę, która jest jedyną komórką mającą wpływ na ruch maszyny Turinga. Początkowo taśma zawiera symbole $a_1a_2...a_nBB...$ gdzie $a_1a_2...a_n$ jest słowem wejściowym z alfabetu wejściowego (podzbiór alfabetu taśmy), natomiast B jest symbolem *pustym*.

Formalna definicja Maszyny Turinga

Maszyna Turinga to siódemka: $MT=(Q, \sum, \Gamma, \delta, q_0, B, F)$, gdzie:

- Q jest skończonym zbiorem stanów
- $\sum$ jest alfabetem wejściowym (alfabet, na bazie którego zbudowane są słowa na wejściu automatu)
- Γ jest alfabetem taśmy; $\Gamma \subseteq \sum$
- $B \in \Gamma - \sum$ jest symbolem pustym

- $q_0 \in Q$ jest stanem startowym maszyny; $F \subseteq Q$ jest zbiorem stanów akceptujących maszyny
- Funkcja przejścia δ pobiera jako argumenty: aktualny stan oraz symbol taśmy wskazywany przez głowicę maszyny i zwraca nowy stan (może być ten sam), zapisuje do komórki taśmy wskazywanej przez głowicę nowy symbol (może być ten sam) oraz podaje kierunek, w którym głowica maszyny się przesunie Lewo / Prawo (L/P).

Przykładowa Maszyna Turinga

Poniżej przedstawiona jest prosta Maszyna Turinga, która akceptuje słowa, których trzecim symbolem jest 0. W przeciwnym wypadku maszyna działa w nieskończoność.

M=({p,q,r,s,t}, {0,1}, {0,1,B}, p, B, {s}), a funkcja δ posiada następujące prawa:

1. $\delta(p, X) = (q, X, R)$ dla $X=0, 1$
2. $\delta(q, X) = (r, X, R)$ dla $X=0, 1$
3. $\delta(r, 0) = (s, 0, L)$
4. $\delta(r, 1) = (t, 1, R)$
5. $\delta(t, X) = (t, X, R)$ dla $X=0, 1, B$

Opis stanu Maszyny Turinga

Opis stanu maszyny Turinga pozwala na jednoznaczne zapisanie tego co dzieje się w każdym momencie działania Maszyny Turinga. Na opis stanu MT składa się: aktualny stan maszyny, zawartość jej taśmy oraz pozycja na taśmie wskazywana przez głowicę maszyny. Możemy sprawić, by opis stanu MT dla nieskończonej taśmy był skończony poprzez pomijanie symbolów na taśmie, które znajdują się jednocześnie po prawej stronie głowicy oraz występują po pierwszym od lewej symbolu B. Mimo, iż nie ma żadnego ograniczenia na to jak daleko w prawo może się przesunąć głowica, to jednak po skończonym czasie MT może odwiedzić tylko skończony prefiks nieskończonej taśmy.

Na opis stanu składa się trójka $\alpha q \beta$, gdzie:

- q jest stanem, w którym aktualnie znajduje się MT
- α jest zawartością taśmy po lewej stronie głowicy maszyny
- β jest niepustą zawartością taśmy po prawej stronie głowicy włączając symbol wskazywany przez głowicę (głowica wskazuje na pierwszy symbol β)

Pojedynczy ruch MT jest oznaczany symbolem $\neg$ i podobnie jak miało to miejsce przy automatach skończonych, notacja $\neg^*$ oznacza domknięcie

przechodnie $\neg$. Możliwe przejścia Maszyny Turinga, dla $\alpha,\beta,a,b \in \Gamma^*$ oraz $q_k \in Q$, $q_m \in Q-F$ to:

1. $\alpha q_k a\beta \neg \alpha q_m b\beta$ zamiana symbolu wskazywanego przez głowę
2. $\alpha q_k a\beta \neg \alpha a q_m \beta$ przesunięcie głowicy w prawo
3. $\alpha a q_k \beta \neg \alpha q_m a\beta$ przesunięcie głowicy w lewo
4. $\alpha q_k B \neg \alpha B q_m B$ wyjście poza zakres symboli niepustych

Dla powyższego zestawu produkcji albo nie ma w ogóle produkcji typu 2, 3, albo są one zdefiniowane dla każdego $a \in \Gamma$.

Opis stanu dla przykładowej Maszyny Turinga

Dla Maszyny Turinga M podanej wcześniej, ciąg opisów stanów dla słowa wejściowego 0101 będzie następującą sekwencją:

p0101 $\neg$ 0q101 $\neg$ 01r01 $\neg$ 0s101, po czym M znajduje się w stanie akceptującym s więc kończy działanie. Dla słowa wejściowego 0111 maszyna M nigdy nie zakończy działania. Sekwencja opisów stanów będzie tutaj następująca:

p0111 $\neg$ 0q111 $\neg$ 01r11 $\neg$ 011t1$\neg$ 0111t $\neg$ 0111Bt $\neg$ ...

Akceptacja przez stan końcowy oraz zatrzymanie

Jednym ze sposobów określenia języka maszyny Turinga jest podanie słów, pod wpływem których maszyna znajdzie się w stanie końcowym. Formalnie:

- $L(M) = \{w \mid q_0 w \neg^* \alpha p \beta$ dla pewnego $p \in F$ oraz dowolnych $\alpha, \beta \in \Gamma^*\}$

Innym sposobem określenia języka maszyny Turinga jest podanie słów, pod wpływem których maszyna się *zatrzyma* tzn. znajdzie się w takiej sytuacji (stan, symbol wskazywany przez głowicę), w której nie będzie zdefiniowany żaden ruch. Formalnie:

- $H(M) = \{w \mid q_0 w \neg^* \alpha p X \beta$ oraz $\delta(p, X)$ nie jest zdefiniowany$\}$
  Czasem traktuje się sytuacje „wypadnięcia głowicy z lewej strony taśmy" jako zatrzymanie maszyny Turinga. Taką maszynę możemy jednak łatwo przekształcić do maszyny akceptującej ten sam język, której głowica zamiast wyskakiwać z lewej strony taśmy przechodzi do stanu, z którego nie ma już wyjścia.

Przykład:

Dla przykładowej maszyny Turinga podanej na początku tej sekcji, język rozpoznawany przez stan końcowy jest równy następującemu wyrażeniu regularnemu:

$L(M) = (0+1)\ (0+1)\ 0\ (0+1)^*$

Dla tej samej maszyny język rozpoznawany przez zatrzymanie jest równy następującemu wyrażeniu regularnemu:

$H(M) = \varepsilon + 0 + 1 + (0+1)(0+1) + (0+1)(0+1)0(0+1)^*$.

Równoważność akceptacji przez stan końcowy oraz zatrzymanie

Można udowodnić następujące twierdzenie: język L jest rozpoznawany przez maszynę M_1 przez stan końcowy wtedy i tylko wtedy gdy jest on rozpoznawany przez zatrzymanie przez pewną maszynę M_2. Formalnie $L=L(M_1) \Leftrightarrow L=H(M_2)$.

Dowód $\Leftarrow$

Należy zmodyfikować maszynę M_2 w następujący sposób:

- Dodać jeden stan końcowy r.
- Dla każdej sytuacji bez wyjścia maszyny M_1 (pary (q,X) takiej że $\delta(q, X)$ nie jest zdefiniowane) dodaj przejście ze stanu q do stanu r bez zmiany X przy jednoczesnym przesunięciu głowicy w prawo (co zapobiega wypadnięciu głowicy z lewej strony taśmy). Teraz kiedykolwiek maszyna M_2 dojdzie do sytuacji „bez wyjścia" będzie mogła znaleźć się w stanie końcowym.

Dowód $\Rightarrow$

Ogólnie rzecz biorąc maszyna M_1 będzie symulowała maszynę M_2. Za każdym razem kiedy maszyna M_2 osiąga stan końcowy, nie ma zdefiniowanych przejść dla tego stanu więc maszyna M_1 automatycznie się zatrzymuje. Pozostaje jednak problem, gdy maszyna się zatrzymuje na słowie, które nie jest akceptowane. W tej sytuacji chcielibyśmy aby maszyna M_1 nie zatrzymywała się, lecz podążała w nieskończoność w prawą stronę. Rozwiązanie takie otrzymujemy przez następującą konstrukcję M_1 na podstawie M_2:

- Dodajemy do M_1 nowy stan r, przy którym następuje przejście w prawo do stanu r pod wpływem dowolnego symbolu bez jego modyfikacji.
- Dla każdej sytuacji bez wyjścia maszyny M_2 (pary (q,X) takiej że $\delta(q, X)$ nie jest zdefiniowane) dodaj przejście ze stanu q do stanu r bez zmiany X przy jednoczesnym przesunięciu głowicy w prawo.
- Usunąć przejścia M_1, które prowadzą ze stanu akceptującego M_1 (w celu zatrzymania maszyny M_2 w stanie akceptującym).

6.3 Rodzaje maszyn Turinga

Często patrzymy na stany i taśmę maszyny Turinga jako na bardziej złożoną strukturę; Segmenty taśmy MT traktujemy jako ścieżki. W zbiorze stanów wyodrębniany jest element kontrolny odpowiedzialny za wykonywanie programu na maszynie Turinga. Poszczególne segmenty taśmy przechowują dane dla wykonywanego programu.

Przykład wymiany symboli

Załóżmy, że mamy maszynę Turinga $M=(Q, \Sigma, \Gamma, \delta, q_0, B, F)$. Niech program M potrzebuje czasami wymienić zawartość sąsiadujących komórek taśmy. Zadanie to może być zrealizowane przez następujący schemat:

- Niektóre spośród stanów Q będą miały postać [q,X] oraz [p,X] dla każdego symbolu taśmy ze zbioru Γ (każdy taki symbol będzie mógł być wymieniony). Będziemy także potrzebowali dodatkowych stanów r oraz s określających początek i koniec procesu wymiany sąsiednich symbolów taśmy.
- W stanie r (początek wymiany) następuje pobranie symbolu taśmy i wstawienie go do opisu stanu związanego z symbolem taśmy (sam symbol na taśmie pozostaje bez zmian). Dodajemy więc do maszyny przejścia:
    - $\delta(r,X) = ([q,X],X, R)$ dla każdego symbolu $X \in \Gamma$, który może być wymieniony
- W stanie kontrolnym [q,X] maszyna wstawia na aktualnie wskazywane miejsce symbol X pobrawszy najpierw z tej komórki taśmy symbol Y. Następnie maszyna przesuwa się w lewo do stanu [p,Y]. Formalnie muszą istnieć przejścia:
    - $\delta([q,X],Y) = ([p,Y], X, L)$ dla wszystkich symboli $X,Y \in \Gamma$, które mogą być wymienione.
- W stanie kontrolnym [p,Y] następuje zapisanie do aktualnie wskazywanej komórki taśmy zawartości stanu [p,Y] związanego z symbolem taśmy. Tym samym maszyna przechodzi do stanu s kończącego proces wymiany sąsiednich symboli taśmy:
    - $\delta([p,Y],X) = (s, Y, R)$ dla wszystkich symboli $X,Y \in \Gamma$, które mogą być wymienione.

Przykład z wieloma ścieżkami

Często taśma maszyny Turinga składa się ze sklejonych równolegle ze sobą wielu ścieżek. Zastosowaniem wielu ścieżek jest przeznaczenie jednej ścieżki na dane, a pozostałych na „znaczniki”. Każdy symbol X alfabetu Γ

zamieniany jest na parę [A,X], gdzie A jest albo puste (B) albo *. Symbol wejściowy a przybiera więc postać [B,a] natomiast symbol pusty postać [B,B].

Takie rozszerzenie symboli alfabetu Γ na dwie równoległe ścieżki pozwala na napisanie prostego programu znajdującego znacznik *, jeśli znajduje się on gdzieś po lewej stronie aktualnej pozycji. Potrzebne są nam następujące elementy funkcji przejścia:

1. $\delta(q, [B,X]) = (q, [B,X], L)$ dla każdego $X \in \Gamma$
2. $\delta(q, [*,X]) = (p, [B,X], R)$ dla każdego $X \in \Gamma$

Drugie przejście oznacza odnalezienie znacznika *.

Mimo, iż języki regularne oraz bez-kontekstowe są klasami języków zdefiniowanymi przy pomocy wygodnych notacji (wyrażenia regularne, gramatyki bez-kontekstowe) nikt nie uważał, że określają one „wszystko co można obliczyć". Maszyna Turinga została więc stworzona jako mechanizm pozwalający na rozwiązanie wszystkiego, co można obliczyć. Jako że jest ona zdefiniowana dla rozpoznawania dowolnego języka, a modele obliczeniowe są zapisywane przy pomocy różnych języków możemy maszynę Turinga traktować jako maszynę *uniwersalną*.

Skoro maszyna Turinga rzeczywiście jest w stanie symulować dowolny ciąg obliczeń, naszym następnym krokiem powinno być pokazanie, że może ona naśladować inne modele obliczeń, w szczególności: wielościeżkową maszynę Turinga, niedeterministyczną maszynę Turinga, maszyny wielo-stosowe, maszyny zliczające i w ostateczności „prawdziwe" komputery.

Wielotaśmowa maszyna Turinga

Definicja

Wielotaśmowa maszyna Turinga posiada k niezależnych taśm i k odpowiadających im głowic. Ruch wielotaśmowej maszyny Turinga zależy od aktualnego stanu oraz od k symboli, na które wskazują kolejne głowice maszyny. Akcja to przejście do nowego stanu, wpisanie różnych symboli w k miejsc wskazywanych przez k głowic i przesunięcie głowic (L – lewo, R – prawo, S – postój). Początkowo pierwsza taśma przechowuje dane wejściowe, natomiast pozostałe taśmy są puste.

Symulacja wielu taśm na jednej taśmie

W celu symulacji k taśm używamy jednej taśmy złożonej ze sklejonych równolegle k ścieżek.

- 2k ścieżek dzielimy na k par po 2 ścieżki
- druga ścieżka i-tej pary przechowuje symbole i-tej taśmy
- pierwsza ścieżka i-tej pary przechowuje znacznik pozycji głowicy na i-tej taśmie:

i-ta	Pierwsza ścieżka				*		
para	Druga ścieżka		E	F	X	Z	

- W celu wykonania ruchu jednotaśmowa maszyna musi:
  - Dla kolejnych k par znaleźć kolejne * przy pomocy których można ustalić na co wskazują głowice w maszynie wielotaśmowej. Takie przeszukiwanie polega na przesuwaniu się w lewo lub prawo w celu znalezienia symbolu * dla kolejnej taśmy. Po znalezieniu * taśmy i należy pobrać symbol wskazywany przez głowicę i-tej taśmy i zapamiętać go w opisie aktualnego stanu
  - Jeśli wyznaczymy opis stanu postaci $[q,s_1,s_2,...s_k]$ to możemy sprawdzić co zrobiłaby maszyna wielotaśmowa dla stanu q i wskazywanych symboli $s_1,s_2,...s_k$. Załóżmy, że przeszłaby do stanu p wstawiając na k taśm symbole $t_1,t_2,...,t_k$.
  - Dla kolejnych k par należy odnaleźć znaczniki * i w miejscach przez nie wskazywanych zamienić odpowiednie s_i na t_i. Na końcu należy przesunąć wszystkie symbole * w prawo lub lewo w zależności od tego, czy wielotaśmowa maszyna w stanie $[q,s_1,s_2,...s_k]$ przesunęła się w prawo, czy lewo.

W przypadku symulacji maszyn wielotaśmowych o wielomianowym czasie obliczeń na maszynie jednotaśmowej można zauważyć ciekawy fakt: jeśli dla danych wejściowych wielkości n maszyna wielotaśmowa wykona T(n) ruchów, to odpowiadająca jej maszyna jednotaśmowa wykona $O(T^2(n))$ ruchów. W dowodzie tego faktu korzysta się z tego, że poszczególne symbole * nie są oddalone o więcej niż T(n) komórek, stąd symulacja jednego ruchu maszyny wielotaśmowej nie zabiera maszynie jednotaśmowej więcej czasu niż O(T(n)).

W konsekwencji jeśli maszyna wielotaśmowa działa w czasie wielomianowym, to odpowiadająca jej maszyna jednotaśmowa także działa w czasie wielomianowym.

Niedeterministyczna maszyna Turinga

Niedeterministyczna maszyna Turinga może mieć więcej niż jedną możliwość przejścia dla aktualnej sytuacji. Przykładowo dla danego stanu q oraz symbolu a wskazywanego przez głowicę funkcja przejścia może posiadać elementy: $\delta(q,a) = (p,b,R)$ oraz $\delta(q,a) = (r,c,L)$. Podobnie jak w przypadku automatów ze stosem nie ma jednak możliwości „przemieszania ruchu" tzn. dla powyższych ruchów nie ma możliwości wykonania ruchu (p,c,R) jeśli nie jest on explicite zadeklarowany.

Symulacja maszyny deterministycznej na maszynie niedeterministycznej nie jest żadnym problemem, pozostaje nam więc zasymulować maszynę niedeterministyczną na maszynie deterministycznej.

Symulacja maszyny niedeterministycznej na maszynie deterministycznej

Oznaczmy niedeterministyczną maszynę Turinga jako NMT, a deterministyczną maszynę Turinga jako DMT. Niech NMT posiada jedną taśmę (która może symulować wielotaśmową NMT). Przeprowadzimy symulację takiej NMT na wielotaśmowej DMT. Później będzie można przekształcić taką wielotaśmową DMT do jednotaśmowej DMT.

- Jedna taśma DMT będzie przechowywać kolejkę opisów stanów NMT. Poszczególne elementy tej kolejki będą oddzielone znacznikiem *.
- Kiedy przykładowy opis stanu ($\alpha qX\beta$) dotrze na przód kolejki, to jeśli NMT posiada przejścia $\delta(q,X) = (q_1,x_1,k_1)$, $\delta(q,X) = (q_2,x_2,k_2)$, ..., $\delta(q,X) = (q_n,x_n,k_n)$ wówczas usuwamy z przodu kolejki opis stanu ($\alpha qX\beta$) i wstawiamy na koniec kolejki opisy stanów (q_1,x_1,k_1), (q_2,x_2,k_2), ..., (q_n,x_n,k_n).
- Akceptacja słowa następuje wtedy, gdy na początku kolejki znajdzie się opis stanu akceptującego.

Porządek kolejki jest bardzo ważny, gdyż pozwala DMT na odwiedzenie wszystkich (opisów) stanów, które odwiedza NMT. Jeśli zamiast kolejki zastosowalibyśmy stos, wówczas NMT mógłby znaleźć się w stanie akceptującym, natomiast symulująca go DMT mogłaby w nieskończoność wpaść w proces poszukiwania stanu akceptującego nie tą drogą co trzeba.

Maszyna wielo-stosowa

Maszyna wielo-stosowa różni się od automatu ze stosem, że może posiadać więcej niż jeden stos.

Automat ze stosem nie może symulować maszyny Turinga, bo wówczas mógłby rozpoznawać języki kontekstowe. Okazuje się jednak, że wielo-stosowa maszyna składająca się z 2 stosów jest w stanie symulować dowolną maszynę Turinga. Pomysł stojący za tą symulacją jest następujący: jeden stos wykorzystujemy na przechowywanie symboli taśmy na lewo od głowicy MT, a drugi stos wykorzystujemy na przechowywanie symboli taśmy na prawo od głowicy MT. Z uwagi na to, że każda (niedeterministyczna, wielo-taśmowa) maszyna Turinga może być sprowadzona do jednotaśmowej DMT, maszyna wielo-stosowa może być używana do symulowania dowolnej maszyny Turinga.

Z drugiej strony, jeśli zawartość stosu rozmieścimy na kolejnych taśmach, wówczas maszynę wielo-stosową może z powodzeniem symulować wielo-taśmowa maszyna Turinga.

Maszyna zliczająca

Są dwie równoważne definicje maszyny zliczającej:

1. Stos ze znacznikiem spodu stosu Z_0 oraz jednym innym symbolem (np. X), który może być dokładany na stos. Stos ma zawsze postać $XX...XZ_0$.
2. Urządzenie przechowujące liczbę naturalną z operacjami: dodaj 1, odejmij 1, sprawdź-czy-zero.

Licznik jest więc czymś w rodzaju czarnej skrzynki, która przechowuje jakąś liczbę naturalną. Możliwości licznika, to jego trzy operacje i aparat logiczny (np. jeśli po odjęciu 1 liczba będzie 0 to dodaj 5 itp.) 1-Licznik przechowuje 1 liczbę, 2-licznik przechowuje 2 liczby, n-licznik przechowuje n liczb. Dla k-licznika możemy dla każdej liczby osobno wykonywać jego operacje.

1-Licznik możemy wykorzystać do rozpoznawania wszystkich języków regularnych i niektórych języków bez-kontekstowych, jak $\{0^n1^n \mid n \in \aleph\}$.

2-Licznik rozpoznaje te same języki co maszyna Turinga. Dowód jest dwuczęściowy: najpierw dowodzimy, że 3-licznik symuluje maszynę 2-stosową, a następnie, że 2-licznik symuluje 3-licznik.

<u>3-Licznik symulujący maszynę 2-stosową</u>

Niech stos maszyny 2-stosowej składa się z *r-1* symboli. Jeśli każdemu symbolowi przyporządkujemy unikatowy numer z przedziału 0..r-1, to zawartość stosu możemy interpretować z liczbą w systemie r; Każda cyfra tej liczby odpowiada jednemu symbolowi na stosie. Symulacja maszyny 2-stosowej na 3 licznikach jest następująca:

- Użyj dwóch liczników na przechowanie liczb reprezentujących zawartość dwóch stosów. Trzeci licznik będzie licznikiem roboczym.
- Zbuduj operacje mnożenia i dzielenia przez r na licznikach:
    - Dzielenie licznika A przez r: odejmij r od A dodając do licznika tymczasowego 1. Powtarzaj czynność dopóki możliwe będzie odjęcie r od A. Wynik znajdzie się w liczniku tymczasowym
    - Mnożenie licznika A przez r: odejmij 1 od A dodając do licznika tymczasowego r. Powtarzaj czynność dopóki A będzie różne od 0. Wynik znajdzie się w liczniku tymczasowym.
- Operacja umieszczenia na stosie A symbolu X: pomnóż liczbę w A przez r, a następnie dodaj do A liczbę reprezentowaną przez cyfrę X.
- Operacja usunięcia symbolu ze stosu A: podziel liczbę w A przez r, a następnie pomiń resztę dzielenia.
- Przeczytaj symbol ze szczytu stosu A: przepisuj zawartość licznika A do licznika tymczasowego partiami po r. Jeśli liczba w A jest mniejsza od r, to jest to element na szczycie stosu

2-Licznik symulujący 3-licznik

Niech liczniki A B C 3-licznika przechowują liczby a, b oraz c. Pierwszy licznik 2-licznika będzie przechowywał liczbę $2^a3^b5^c$, drugi licznik 2-licznika będzie pomocniczy.

- Test, czy licznik A jest równy 0 wykonywany jest w następujący sposób: wartość z licznika A przepisywana jest partiami 2 elementowymi do licznika tymczasowego. Jeśli w liczniku A pozostanie liczba większa od 0 oznacza to, że $2^a3^b5^c$ nie jest podzielna przez 2, czyli licznik A jest równy 0.
- Testy, czy liczniki B oraz C są zerowe są analogiczne (przepisujemy liczbę z B i C odpowiednio po 3 i 5 elementów).
- Dodawanie 1 do A, B, C polega na mnożeniu liczby w 2-liczniku odpowiednio przez 2, 3 i 5.
- Odejmowanie 1 od A, B, C polega na dzieleniu liczby w 2-liczniku odpowiednio przez 2, 3 i 5.

Prawdziwy komputer

Komputer jest słabszym modelem obliczeniowym niż maszyna Turinga, gdyż jego ograniczona ilość stanów i danych wejściowych nigdy nie zastąpi nieskończonej taśmy maszyny Turinga. Chcąc symulować maszynę Turinga na komputerze potrzebowalibyśmy nieskończonej ilości fizycznej pamięci.

7. Funkcje rekurencyjne

Streszczenie: *W rozdziale tym przyjrzymy się funkcjom rekurencyjnym, na bazie których zbudowane są programy komputerowe. Na początku przedstawimy rekurencję prymitywną, oraz schemat rekurencji prymitywnej, dzięki któremu można wychodząc z funkcji bazowych: zera, następnika oraz projekcji wygenerować nowe funkcje rekurencyjnie prymitywne. W dalszej części rozdziału wprowadzimy także pojęcie rekurencji nie-prymitywnej definiującej funkcje bardziej złożone, jak np. funkcja Ackermanna. W rozdziale przedstawiony zostanie także sposób kodowania ciągów, który redukuje problemy opierające się na skończonej ilości zmiennych do problemów opierających się na jednej zmiennej.*

7.1 Teza Churcha

Czy wszystkie funkcje obliczalne są rekurencyjne? Na to pytanie można odpowiedzieć tylko dzięki dokładnej definicji natury funkcji obliczalnej. *Teza Church'a* (1936) utrzymuje, że funkcje obliczalne są dokładnie funkcjami rekurencyjnymi. Doświadczenia pokazują, że za każdym razem, gdy funkcja jest zdefiniowana przy pomocy efektywnej procedury (algorytmu) , procedura ta dostarcza środków, by dowieść, że funkcja jest w rezultacie rekurencyjna.

Niemniej jednak, teza Churcha nie mówi nic o reprezentacji algorytmu związanego z tą funkcją rekurencyjną. Z punktu widzenia informatyka, wielkości takie jak czas oraz wielkość pamięci potrzebnej do obliczeń zależą od modelu rozpatrywanej maszyny. Z drugiej strony, wszystkie modele przyzwoitych maszyn definiują taką samą klasę funkcji: funkcji rekurencyjnie częściowych.

7.2 Rodzaje funkcji rekurencyjnych

Aby zdać sobie sprawę z wszystkich funkcji obliczalnych, konieczne jest zdefiniowanie klasy większej niż klasa funkcji rekurencyjnie prymitywnych. Może to zostać osiągnięte dzięki nowemu schematowi konstrukcji nazywanemu *schematem minimalizacji*.

Wprowadzenie do tego schematu pociąga za sobą konieczność skupienia się na funkcjach częściowych. Dla danego podzbioru A przestrzeni N^{p+1} chcemy zdefiniować funkcję p argumentową, która z ciągiem liczb $(x_1, x_2, ..., x_p)$ wiąże najmniejszą wartość „z", taką że $(x_1, x_2, ..., x_p, z) \in A$. Jeśli wystąpi przypadek, że dla każdej wartości zmiennej „z" ciąg $(x_1, x_2, ..., x_p, z) \notin A$ w przeciwieństwie do minimalizacji ograniczonej, schemat minimalizacji nie zatrzymuje się.

Definicje:

Funkcją częściową z przestrzeni N^p w N nazywamy parę (D,f) gdzie D jest podzbiorem przestrzeni N^p ,a f funkcją ze zbioru D w N. D nazywamy dziedziną definicji funkcji f. W przypadku, gdy $D = N^p$, f nazywamy funkcją *totalną*.

Jeżeli $(x_1, ..., x_p) \in D$, wówczas f jest zdefiniowana dla $(x_1, ..., x_p)$ co oznaczamy $f(x_1, ..., x_p)\downarrow$. Jeśli $(x_1, ..., x_p) \notin D$, wtedy f nie jest zdefiniowana dla $(x_1, ..., x_p)$.

Należy zauważyć, że dwie funkcje częściowe są równe, jeżeli posiadają taki sam zbiór definicji i jeśli są identyczne na jego dziedzinie, trzeba więc przy każdej nowo tworzonej funkcji dokładnie określić jego dziedzinę definicji:

1. Dziedzina funkcji f , złożonej z „p" funkcji n - argumentowych $g_1, .., g_p$ oraz p – argumentowej funkcji h jest definiowana przez warunki:

$f(x_1, ..., x_p)\downarrow$

wtedy i tylko wtedy gdy:

$g_1(x_1, ..., x_p)\downarrow$

.....

$g_p(x_1, ..., x_p)\downarrow$

$h(g_1(x_1, ..., x_p), ..., g_p(x_1, ..., x_p))\downarrow$

2. Dziedzina funkcji f definiowanej przez rekurencję prymitywną przy pomocy p – argumentowej funkcji g oraz p+2 argumentowej funkcji h jest definiowana przez rekurencję:

$f(x_1, ..., x_p, 0)\downarrow$ wtw $g(x_1, ..., x_p)\downarrow$

$f(x_1, ..., x_p, y+1)\downarrow$ wtw $h(x_1, ..., x_p, y, f(x_1, ..., x_p, y))\downarrow$

Schemat minimalizacji pozwala na skonstruowanie nowych funkcji częściowych:

Weźmy p+1 argumentową funkcję g. Mówimy, że funkcja częściowa f jest zdefiniowana przez minimalizację począwszy od g, gdy:

- istnieje co najmniej jedna zmienna z , taka że $g(x_1,...,x_p, z) = 0$ i jeśli dla każdego $z'<z$ $g(x_1,...,x_p, z') \neq 0$,wtedy $f(x_1,...,x_p) = z$;

w przeciwnym przypadku, $f(x_1,...,x_p)$ nie jest zdefiniowana.

Definicje:

Zbiór *funkcji rekurencyjnych częściowych* jest najmniejszym zbiorem funkcji zawierającym funkcje bazowe i domknięty przez złożenie, rekurencje prymitywną i minimalizację.

Podzbiór przestrzeni N^p nazywamy rekurencyjnym, jeśli jego funkcja charakterystyczna jest rekurencyjna.

Funkcje rekurencyjne częściowe są przeliczalne w takim sensie, że dla każdej z nich istnieje algorytm, który albo zatrzyma się pod koniec ograniczonego czasu zwracając wartość funkcji, jeśli jest ona zdefiniowana albo w przeciwnym przypadku nie zatrzyma się.

7.3 Funkcje rekurencyjnie prymitywne

Funkcjami rekurencyjnie prymitywnymi nazywamy te funkcje, które analizuje bezpośrednio komputer. Konstrukcja układów scalonych pozwala na wygenerowanie podstawowych funkcji, czyli tzw. *funkcji bazowych*, wśród których wyróżniamy:

- funkcję *stałą* z N w N, której wartość stale jest równa 0.
- funkcję *następnika* oznaczaną *suc*, która ze zmienną wejściową x wiąże wartość x+1.
- funkcję projekcji pr_p^i przestrzeni N^p w N zdefiniowaną jako $pr_p^i(x_1,...x_p) = x_i$ dla i=1, 2, ..., p.

Definiujemy *f* jako *funkcję złożoną* z funkcji g_1, g_2, ..., g_p odwzorowujących przestrzeń N^n w N oraz funkcji *h* z przestrzeni N^p w N w następujący sposób:

$f(x_1, ..., x_n) = h[g_1(x_1, ..., x_n), ..., g_p(x_1, ..., x_n)]$

Schemat rekurencji prymitywnej wiąże z dwiema funkcjami *g* oraz *h* mającymi odpowiednio p oraz p+2 argumentów, funkcję p+1 argumentową *f* zdefiniowaną w sposób następujący:

- $f(x_1, ..., x_p,0) = g(x_1, \ldots, x_p)$
- $f(x_1, \ldots, x_p, y+1) = h[x_1, \ldots, x_p, y, f(x_1, \ldots, x_p,y)]$

W powyższym przypadku funkcja f została zdefiniowana przez *rekurencję* przy użyciu funkcji inicjującej g oraz funkcji h będącej etapem rekurencji.

Przykłady:

Przy użyciu rekurencji i złożenia możemy, korzystając z funkcji projekcji i następnika skonstruować funkcję:

Dodawania (a+b) oznaczmy jako +(a,b):

$$+(x,0) = pr_1^1(x)$$
$$+(x,y+1) = pr_3^3(x,y,+(x,y)) + 1$$

Mnożenia (a*b) oznaczamy jako *(a,b):

$$*(x,0) = C_0$$
$$*(x,y+1) = pr_3^3(x,y, *(x,y)) + x$$

gdzie C_0 jest funkcją stale równą 0.

Poprzednika liczby:

$$pred(0) = C_0$$
$$pred(k+1) = pr_2^1(k, pred(k))$$

gdzie $C_1 = succ(C_0)$

Zera, która dla zera przyjmuje wartość 1, a dla każdej innej liczby 0:

$$zero(0) = C_1$$
$$zero(k+1) = pred[pr_2^2(k, zero(k))]$$

Odejmowania symetrycznego:

$$n \div 0 = 0$$
$$n \div (k+1) = pred(pr_3^3(n, k, n \div k))$$

Definicje:

- zbiór T funkcji jest domknięty ze względu na proces konstrukcji, jeśli każda funkcja f zdefiniowana przy pomocy funkcji z T przy pomocy tego procesu jest także w T.
- zbiór *funkcji rekurencyjnie prymitywnych* jest najmniejszym spośród zbiorów funkcyjnych zawierających funkcje bazowe i jest on domknięty przez złożenie i rekurencję prymitywną.
- podzbiór A przestrzeni N^p nazywamy *rekurencyjnie prymitywnym*, jeśli jego *funkcja charakterystyczna* jest rekurencyjnie prymitywna.

Funkcją charakterystyczną podzbioru A przestrzeni N jest:

$$\chi_A(x) = \begin{cases} 1 \text{ gdy } x \in A \\ 0 \text{ w przeciwnym przypadku} \end{cases}$$

- Relacja R oparta na ciągach długości p jest *rekurencyjnie prymitywna*, jeśli zbiór $\{(x_1, ..., x_p) \in N^p : R(x_1, ..., x_p)\}$ jest rekurencyjnie prymitywny.

Aby pokazać, że funkcja jest rekurencyjnie prymitywna wystarczy sprawdzić czy może ona zostać otrzymana począwszy od funkcji bazowych przy wykorzystaniu złożeń i schematu rekurencji prymitywnej.

Przykład podzbiorów rekurencyjnie prymitywnych:

Każdy jednoelementowy podzbiór (np. {m}) zbioru liczb naturalnych jest rekurencyjnie prymitywny, gdyż możemy dla niego skonstruować następującą funkcję charakterystyczną, która będąc złożeniem funkcji prymitywnych, także będzie prymitywna:

$\chi_{\{m\}}(x) = \text{zero}\{ +(n \div x, x \div n)\}$

Widzimy, że jeśli $n \neq x$, wtedy funkcja + zwróci wartość większą od 0, co w wyniku działania funkcji zero da nam wartość 0. Jeśli $n = x$, wówczas funkcja charakterystyczna zwróci 1.

7.4 Konstrukcja funkcji rekurencyjnie prymitywnych

Funkcje rekurencyjnie prymitywne możemy tworzyć na kilka sposobów:

Procesem najczęściej używanym w programowaniu jest *definiowanie przez przypadki*. Zakładamy, że mamy dane dwie p parametrowe funkcje rekurencyjnie prymitywne, oraz podzbiór rekurencyjnie prymitywny A przestrzeni N^p. Zdefiniowana poniżej funkcja h:

$$h(x_1, ..., x_p) = \begin{cases} f(x_1, ..., x_p) \text{ gdy } (x_1, ..., x_p) \in A \\ g(x_1, ..., x_p) \text{ w przeciwnym przypadku} \end{cases}$$

jest rekurencyjnie prymitywna.

W rzeczywistości jeśli weźmiemy χ_A oraz $\chi_{C(A)}$, funkcje charakterystyczne odpowiednio zbiorów A oraz jego dopełnienia C(A), to funkcja h może się wyrażać jako $h = f \bullet \chi_A + g \bullet \chi_{C(A)}$. Biorąc pod uwagę fakt że jest to suma, której składniki są iloczynami funkcji rekurencyjnie prymitywnych dostajemy funkcję rekurencyjnie prymitywną h.

Funkcją rekurencyjnie prymitywną jest suma i iloczyn graniczny wartości funkcji rekurencyjnie prymitywnych. Dla danej p+1 parametrycznej funkcji rekurencyjnie prymitywnej f, funkcje g oraz h zdefiniowane jako:

$$g(x_1, ..., x_p, y) = \sum_{t=0}^{y} f(x_1, \ldots, x_p, t)$$

$$h(x_1, ..., x_p, y) = \prod_{t=0}^{y} f(x_1, \ldots, x_p, t)$$

są także rekurencyjnie prymitywne. Dla przykładu pokażemy, że g może być zdefiniowana przez rekurencję:

$g(x_1, ..., x_p, 0) = f(x_1, ..., x_p, 0)$

$g(x_1, \ldots, x_p, y+1) = g(x_1, \ldots, x_p, y) + f(x_1, \ldots, x_p, y+1)$

Zbiór relacji rekurencyjnie prymitywnych jest domknięty przez *kwantyfikację ograniczoną*, tzn. jeśli A jest podzbiorem rekurencyjnie prymitywnym przestrzeni N^{p+1}, wówczas rekurencyjnie prymitywnymi są także zbiory B i C zdefiniowane jako:

$B = \{(x_1, ..., x_p, z) : \exists\, t \le z \;\; (x_1, ..., x_p, t) \in A\}$

$C = \{(x_1, ..., x_p, z) : \forall\, t \le z \;\; (x_1, ..., x_p, t) \in A\}$

ponieważ możemy dla nich zdefiniować funkcje charakterystyczne w następujący sposób:

$$\lambda_B(x1, ..., xp, z) = sg(\sum_{t=0}^{z} \chi_A(x_1, ..., x_p, t))$$

$$\lambda_C(x1, ..., xp, z) = \prod_{t=0}^{z} \chi_A(x_1, ..., x_p, t)$$

gdzie funkcja sg(x), taka że $sg(0) = 0$ i $sg(x) = 1$ gdy $x \neq 0$ jest także rekurencyjnie prymitywna, bowiem definiujemy ją jako:

$sg(x) = 1 \div zero(x)$

Zatem powyższe funkcje charakterystyczne zbiorów B i C są odpowiednio sg z sumy granicznej funkcji rekurencyjnie prymitywnych oraz iloczynu granicznego funkcji rekurencyjnie prymitywnych.

Do zdefiniowania funkcji rekurencyjnie prymitywnych możemy także użyć *schematu minimalizacji ograniczonej*. Niech zbiór rekurencyjnie prymitywny A będzie podzbiorem przestrzeni N^{p+1}. Zdefiniowana poniżej funkcja f jest także rekurencyjnie prymitywna:

$$f(x_1, ..., x_p, z) = \begin{cases} t \text{ ,najmniejsza wartość } \le z \\ \text{spełniająca } f(x_1, ..., x_p, t) \in A \\ 0 \text{ w przeciwnym przypadku} \end{cases}$$

Jest to funkcja oznaczana niekiedy jako:

$f(x_1, ..., x_p,z) = \mu t \leq z \ (x_1, ..., x_p, t) \in A$

Pokażemy, że funkcja f może być zdefiniowana przez rekurencję przy pomocy poprzedniego schematu, definicji przez przypadki oraz sumy ograniczonej:

$f(x_1, ..., x_p, 0) = 0$

$f(x_1, \ldots, x_p, z+1) = f(x_1, \ldots, x_p, z)$ gdy $\sum_{t=0}^{z} \chi_A(x_1, ..., x_p, t) \geq 1$

- ponieważ funkcja charakterystyczna relacji $\geq$ jest także rekurencyjnie prymitywna
- w przeciwnym wypadku ,gdy zachodzi $(x_1, ..., x_p, z+1) \in A$ mamy $f(x_1, \ldots, x_p, z+1) = z+1$
- w każdym innym przypadku jest $f(x_1, \ldots, x_p, z+1) = 0$

Przykłady:

Zbiory skończone

Udowodniliśmy już, że każdy jednoelementowy podzbiór przestrzeni N jest rekurencyjnie prymitywny. Teraz korzystając z sumy ograniczonej udowodnimy, że każdy skończony podzbiór przestrzeni N jest rekurencyjnie prymitywny. Mamy więc dany $A = \{m_1, ..., m_n\} \subset N$. Dla każdego m_i i=1..n mamy już zdefiniowaną funkcję charakterystyczną χ_A^i. Dla zbioru A określamy więc funkcję charakterystyczną, jako:

$\chi_A(x) = \sum_{i=0}^{n} \chi_A^i(x)$

będącą funkcją rekurencyjnie prymitywną.

Podobnie dowodzimy, że jednoelementowy podzbiór A przestrzeni N^p jest rekurencyjnie prymitywny. Ostatecznie dla skończonego podzbioru A przestrzeni N^p twierdzimy, że jest on rekurencyjnie prymitywny przez zdefiniowanie dla niego funkcji charakterystycznej. Niech $A = \{c_1, ..., c_m\} \subset N^p$ gdzie kolejne $c_i = (x_{i1}, ... ,x_{ip})$. Dla każdego c_i mamy funkcję charakterystyczną λ_A^i, stąd poszukiwana funkcja charakterystyczna (dla wejściowego ciągu c) będzie miała postać:

$\Psi_A(c) = \sum_{i=0}^{m} \lambda_A^i(c)$

co można również zapisać jako:

$$\Psi_A(x_1, \ldots, x_p) = \sum_{i=0}^{m} \{ \prod_{j=1}^{p} \chi_A^j (pr_p^j (x_1, \ldots, x_p)) \}$$

Zbiory nieskończone

Udowodnimy, że dla danej liczby naturalnej m > 1 , zbiór potęgowy A={1, m, m^2, m^3 ...} jest rekurencyjnie prymitywny. Na początku zauważmy, że zachodzi zależność x < m^x. Zdefiniujemy funkcję charakterystyczną częściową F, która będzie „akceptowała" skończoną ilość potęg m, tzn.:

$F(x,k) = \chi\{1,m^1,m^2, ..., m^k\}$ (x)

Konstruujemy funkcję

$$\chi(a,x) = \begin{cases} 1 \text{ gdy } x=a \\ 0 \text{ w przeciwnym przypadku} \end{cases}$$

w następujący sposób:

$\chi(a,x)$ = zero(+(a÷x, x÷a))

jest to funkcja rekurencyjnie prymitywna zależna od 2 argumentów.

Funkcję F zdefiniujemy rekurencyjnie tak, by w nieskończoności akceptowała ona wszystkie potęgi m.

$F(x,0) = \chi(C_1(x),x)$

Krok inicjujący zależny jest tylko od x.

$F(x,k+1) = pr_3^3(x,k,F(x,k)) + \chi(exp_m(succ(k)), x)$

Krok rekurencyjny zależny jest tylko od x oraz k

Ostatecznie funkcja charakterystyczna dla zbioru A przyjmie postać:

$\Im(x) = F(x,x)$

ponieważ element x < m^x będzie mógł być zaakceptowany przez jedną z funkcji F.

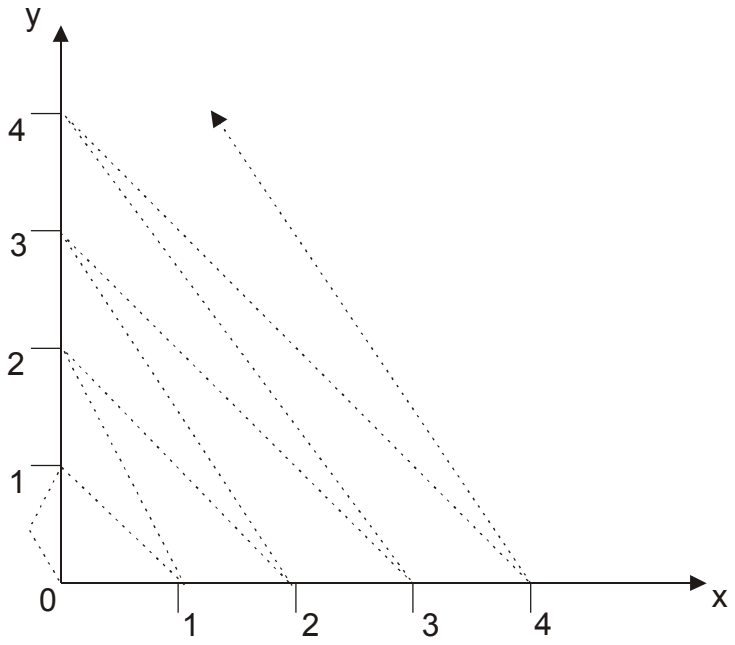
7.5 Kodowanie ciągów

Umiejętność zredukowania problemu opierającego się na dwóch, lub skończonej ilości zmiennych do problemu jednej zmiennej jest w informatyce bardzo użyteczna. Redukcja taka pozwala na zapisanie skończonego ciągu liczb przy pomocy jednej liczby oraz odwrotnie: przywrócenie ciągu liczb na podstawie jednej liczby. W tym podrozdziale udowodnimy istnienie bijekcji pomiędzy przestrzenią N a N^p .

Zaczniemy od znalezienia odwzorowania wzajemnie jednoznacznego pomiędzy przestrzenią N i N^2. Rozpatrzmy funkcję:

$\propto_2(x,y) = (x+y)(x+y+1)/2 + x$

jest to funkcja rekurencyjnie prymitywna. Odpowiada ona procesowi numerowania punktów płaszczyzny o współrzędnych naturalnych kolejnymi liczbami naturalnymi, co przedstawia rysunek nr 15.



Rysunek 15 Pokrycie płaszczyzny kolejnymi liczbami naturalnymi

Udowodnimy na początku, że dla dowolnych dwóch różnych punktów: (a,b) i (c,d) funkcja przyjmuje różne wartości:

Weźmy przypadek, gdy a+b = c+d, czyli obydwa punkty leżą na jakiejś prostej y=-x+w , w≥0 wówczas gdyby:

$\propto_2(a,b) = \propto_2(c,d)$

to:

$(a+b)(a+b+1)/2 + a = (c+d)(c+d+1)/2 + c$

czyli:

a=c oraz b=d, zatem punkty są takie same – sprzeczność.

Weźmy przypadek, gdy $a+b < c+d$ i oznaczmy $a+b=r_1$ $c+d = r_2$.

$\propto_2(a,b) = (a+b)(a+b+1)/2 + a = r_1(r_1+1)/2 + a \le r_1(r_1+1)/2 + {}_r1 = 1+2+ .. +r_1 + r_1 < 1+2+\ldots+r_2 \le r_2(r_2+1)/2 +c = (c+d)(c+d+1)/2 + c = \propto_2(c,d)$

ostatecznie $\propto_2(a,b) < \propto_2(c,d)$

Podobny rezultat dostajemy, gdy a+b > c+d , czyli funkcja $\propto_2$ jest różnowartościowa.

Pozostaje nam jeszcze udowodnić, że dla każdej liczby naturalnej n zdołamy znaleźć parę liczb naturalnych x i y, aby spełnione było równanie:

$\propto_2(x,y) = n$

Dla każdej liczby n możemy tak dobrać liczby r_1 i r_2, by n znalazło się w środku przedziału $[1+2+ ... + r_1, 1+2+ ... +r_2]$, innymi słowy zachodzi:

$r_1(r_1+1) \le n \le r_2(r_2+1)$,

możemy więc dobrać takie δ by było spełnione równanie:

$r_1(r_1+1) + \delta = n$.

Aby ze zmiennych r_1 i δ przejść do poszukiwanych x i y wystarczy rozwiązać układ równań:

$$\begin{cases} x + y = r_1 \\ x = \delta \end{cases}$$

co daje nam poszukiwaną parę (x,y).

Możemy teraz naszą bijekcję rozszerzyć, by odwzorowywała przestrzeń N^p w N. W tym celu definiujemy przez rekurencję $\propto_p$ dla p>2 jako:

$\propto_{p+1}(x_1, ..., x_p, x_{p+1}) = \propto_p(x_1, \ldots, x_{p-1}, \propto_2(x_p, x_{p+1}))$

Tak zdefiniowana funkcja jest bijekcją, co dowieść możemy indukcyjnie.

Pierwszy krok indukcyjny jest spełniony, gdyż bijekcję N^2 w N funkcji $\propto_2(x,y)$ już udowodniliśmy.

Załóżmy teraz, że $\propto_p$ jest bijekcją i spróbujmy dowieść, że $\propto_{p+1}$ jest bijekcją.

Iniekcja: dowód nie-wprost. Weźmy dwa różne ciągi liczb: $c_1=(x_1,x_2, ..., x_p, x_{p+1})$ oraz $c_2=(y_1, y_2, ..., y_p, y_{p+1})$. Jeśli $\propto_{p+1}(c_1) = \propto_{p+1}(c_2)$ wówczas :

$\propto_p(x_1, \ldots, x_{p-1}, \propto_2(x_p, x_{p+1})) = \propto_p(y_1, \ldots, y_{p-1}, \propto_2(y_p, y_{p+1}))$

Wykorzystując fakt, iż $\propto_p$ jest bijekcją otrzymujemy, że $c_1=c_2$ co jest sprzeczne z założeniem, czyli $\propto_{p+1}$ jest iniekcją:

Suriekcja: wynika prosto z definicji rekurencji. Dla dowolnej liczby naturalnej „n” z założenia indukcyjnego istnieją takie $x_1, x_2, ..., x_{p-1}, \delta$, że zachodzi:

$n = \propto_p(x_1, \ldots, x_{p-1}, \delta)$

oraz dla dowolnego δ mamy istnienie takich x_p i x_{p+1}, że ostatecznie:

$n = \propto_p(x_1, \ldots, x_{p-1}, \propto_2(x_p, x_{p+1}))$, czyli

$n = \propto_{p+1}(x_1, \ldots, x_p, x_{p+1}))$ co kończy dowód.

7.6 Rekurencja nie prymitywna

Funkcje rekurencyjnie prymitywne pozwalają nam zaspokoić większość potrzeb przy definiowaniu funkcji w programowaniu, niemniej jednak niektóre programy nie obliczają tych funkcji, ponieważ nie zawsze się kończą: w rzeczywistości obliczają one funkcje zdefiniowane częściowo.

Funkcje obliczalne, głównie definiowalne i nie rekurencyjnie prymitywnie są wyjątkowo rzadkie i w praktyce nie używane. Za przykład posłuży nam *funkcja Ackermanna*. Poniższa funkcja oznaczona przez A jest trudniejszym wariantem tej, która została zdefiniowana przez Ackermanna.

$A(0, x) = x + 2$ dla każdego x
$A(1,0) = 0$ oraz $A(y, 0) = 1$ dla każdego $y \geq 2$
$A(y + 1, x + 1) = A(y, A(y+1, x))$ dla każdego x oraz y.

Dla każdej wartości n, funkcja wiążąca z x wartość A(n,x) jest oznaczana jako A_n. Funkcja ta jest zdefiniowana przez rekurencję:
$A_0(x) = x + 2$, $A_1(x) = 2x$, $A_2(x) = 2^x$ oraz dla każdego $n > 2$:

$A_n(0) = 1$
$A_n(x+1) = A_{n-1}(A_n(x))$.

Dla każdej zmiennej n, funkcja A_n jest więc rekurencyjnie prymitywna. Aby wykazać, że funkcja A nie jest rekurencyjnie prymitywna musimy przeanalizować kolejno:
dla każdego $n > 1$ oraz każdego x, $A(n,x) > x$
kolejne funkcje A_n są ściśle rosnące
jeśli $n \geq 2$, wtedy $A(n, x) \geq A(n-1, x)$
jeśli $x \geq 4$, wtedy $A(n+1, x) \geq A(n, x+1)$
jeśli $n_1, n_2, ..., n_p$ są zmiennymi, wówczas istnieje taka zmienna m, że dla każdego x zachodzi:

$$\sum_{i=1}^{p} A(n_i , x) \leq A(m , x)$$

jeśli f jest funkcją rekurencyjnie prymitywną p zmiennych, wtedy istnieje takie m, że dla każdego $n_1, ..., n_p$ zachodzi:

$$f(n_1, ..., n_p) \leq A(m , \sum_{i=1}^{p} n_i)$$

patrząc na funkcję g zdefiniowaną jako $g(n) = A(n,n)$ wnioskujemy, że funkcja A nie jest rekurencyjnie prymitywna.

8. Obliczalność i złożoność obliczeniowa

Streszczenie: *Na początku tego rozdziału przeanalizujemy problem obliczalności i maszyny Turinga: udowodnimy, że maszyna Turinga może symulować rekurencję prymitywną, oraz że każda funkcja obliczalna przez maszynę Turinga jest rekurencyjna. W dalszej części rozdziału pokażemy szkic rachunku lambda - modelu obliczeń stosowanego w językach funkcyjnych. Na końcu rozdziału przedstawimy problem złożoności obliczeniowej: zdefiniujemy klasy złożoności obliczeniowej określane przez funkcje $\Theta(n)$, $O(n)$, $\Omega(n)$ powiemy co to jest NP-zupełność oraz zademonstrujemy jak udowadnia się, że problem jest NP-zupełny.*

8.1 Obliczalność a maszyna Turinga

Będziemy korzystali z maszyn Turinga posiadających dwie taśmy oraz poniższe właściwości:

- zbiorem symboli używanych jest $\Sigma = \{0,1, b\}$,gdzie symbol b oznacza „blanc" (pusty)
- zbiór stanów Q jest skończony i zawiera dwa stany wyróżnione: stan początkowy q_0 i stan końcowy q_1.
- funkcja przejścia δ jest odwzorowaniem zbioru $Q \times \Sigma^2$ w zbiór $Q \times \Sigma^2 \times \{L, S, R\}^2$, gdzie L reprezentuje przejście w lewo, R reprezentuje przejście w prawo, a S brak ruchu głowicy czytającej.

Weźmy f, p – argumentową funkcję częściową oraz M – maszynę Turinga. W tym rozdziale będziemy badali funkcje M obliczalne – zdefiniowane przy pomocy maszyny Turinga, innymi słowy Turing-obliczalne.

Standardowa reprezentacja zmiennych w maszynie jest reprezentacją binarną. Od tej chwili będziemy posługiwali się inną notacją, która uprości demonstrację: zmienna o wartości 0 jest reprezentowana przez 0, natomiast zmienna „n" jest reprezentowana przez n pozycji, na których znajdują się wartości 1.

W rozpatrywanych przez nas maszynach Turinga pierwsza taśma reprezentuje ciąg danych wejściowych, natomiast druga taśma ciąg będący rezultatem funkcji reprezentowanej przez maszynę.

Na początku udowodnimy, że funkcje bazowe są Turing – obliczalne.

- Funkcja stale równa 0 jest obliczalna na następującej maszynie Turinga:

$$\delta(q_0, 0, b) = (q_0, 0, 0, R, R)$$
$$\delta(q_0, 1, b) = (q_0, 1, 0, R, R)$$
$$\delta(q_0, b, b) = (q_1, b, b, S, S)$$

- Funkcja następnika jest realizowana na maszynie Turinga, której funkcja przejścia spełnia warunki:
$$\delta(q_0, 1, b) = (q_0, 1, 1, R, R)$$
$$\delta(q_0, b, b) = (q_1, b, 1, S, S)$$
- Funkcja projekcji pr_p^i $(1 \leq i \leq p)$ jest obliczalna na maszynie posiadającej p+1 stanów q_0, q_1, q_2, ..., q_p. Wejściem jest ciąg wartości oddzielonych symbolem b. Przykładowo, funkcja pr_2^2 jest obliczalna na 3 – stanowej q_0, q_1, q_2 maszynie, której funkcja przejścia spełnia:
$$\delta(q_0, 1, b) = (q_0, 1, b, R, S)$$
$$\delta(q_0, 0, b) = (q_0, 0, b, R, S)$$
$$\delta(q_0, b, b) = (q_2, b, b, R, S)$$
$$\delta(q_2, 1, b) = (q_2, 1, 1, R, R)$$
$$\delta(q_2, 0, b) = (q_2, 0, 0, R, R)$$
$$\delta(q_2, b, b) = (q_1, b, b, S, S)$$

Pozostaje nam dowieść, że zbiór funkcji częściowych, obliczalnych na maszynie Turinga jest domknięty przez złożenie, rekurencję prymitywną i minimalizację. Rezultat ten możemy osiągnąć kolejno utożsamiając maszynę z każdym procesem konstrukcji funkcji rekurencyjnej.

8.2 Maszyna Turinga jako model obliczeń

Poniższe obliczenia zapewniają, że klasa funkcji rekurencyjnych częściowych zamyka się razem z klasą funkcji Turing – obliczalnych. Przedstawiona demonstracja używa kodowania zbioru operacji wykonanych przez maszynę w trakcie obliczeń.

Wniosek 1:

Każda funkcja częściowa, obliczalna przez maszynę Turinga jest rekurencyjna.

Weźmy f, funkcję częściową, obliczalną przy pomocy maszyny Turinga M, która posiada 2 taśmy i m stanów. Aby pokazać, że funkcja f jest rekurencyjna musimy na początku zakodować sytuację maszyny przy pomocy zmiennej wejściowej t oraz pokazać, że kod jest funkcją rekurencyjnie prymitywną, zależną od t i od warunków początkowych. Każdy stan maszyny „q_i" jest kodowany przez wartość „i", symbol pusty (blanc) przez wartość 0, natomiast symbol „0" przez wartość 2.

Definicje:

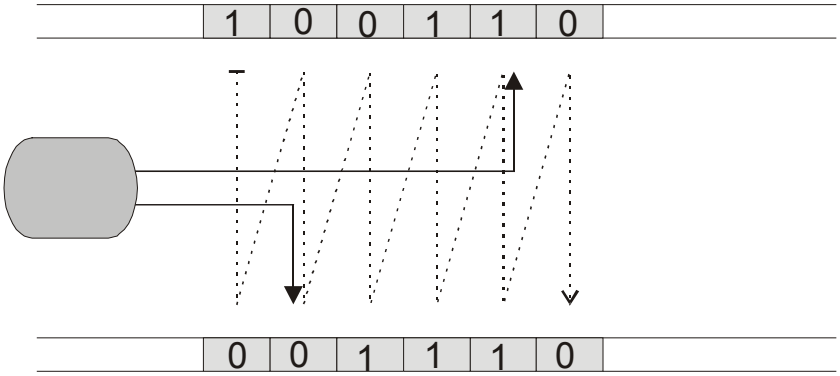
Konfiguracja maszyny M do danego momentu „t" jest nieskończonym ciągiem $C(t) = (s_0, ..., s_i, ...)$ symboli zapisanych do tego momentu przez dwie taśmy maszyny M. Ciąg ten jest uzyskany przez konkatenację ciągów $\sigma_0, \sigma_1, \sigma_2, ..., \sigma_j ...$, gdzie dla każdej wartości „j" , σ_j jest ciągiem symboli zapisanych w komórkach o numerze j. Ten nieskończony ciąg posiada tylko jedną, skończona ilość znaków nie pustych.

Sytuacja maszyny do momentu „t" jest ciągiem $(e, k_1, k_2, C(t))$, gdzie „e" jest kodem stanu maszyny do momentu „t" , k_1 oraz k_2 są numerami komórek, przed którymi do tego momentu znajdują się głowice czytające, a C(t) jest konfiguracją maszyny.

Konfiguracja C(t) może zostać zakodowana przez wartość: $\Gamma(C) = \Sigma_{i\geq 0}\ s_i \bullet 3^i$. Funkcje dzielnika „q" i reszty z dzielenia „r" pozwalają na odzyskanie z powyższego kodu symboli zapisanych w komórce o numerze „u" dla taśmy o numerze „v": $r(q(\Gamma(C), 3^{2(u-1)+v-1}), 3)$. Sytuacja maszyny do momentu „t" może więc zostać zakodowana przez wartość:

$\Gamma(S) = \propto_4(e, k_1, k_2, \Gamma(C))$.

Na rysunku nr 16 przedstawiony jest sposób kodowania maszyny Turinga. Zapisana jest uporządkowana sekwencja C(t) = 1,0,0,0,0,1,1,1,1,1,0,0, k_1=5 k_2=2.



Rysunek 16 Kodowanie maszyny Turinga

Lemat 1

Istnieje funkcja rekurencyjniee prymitywna g, która dostarcza kod sytuacji maszyny do momentu t+1, na podstawie kodu sytuacji maszyny z momentu t.

Dowód:

Przejście zmiennej opisującej w stan następny odbywa się przy pomocy funkcji przejścia. Funkcja, która pozwala wyrazić konfigurację maszyny do momentu „t+1" przy pomocy konfiguracji do momentu „t" może być zdefiniowana w sposób rekurencyjnie prymitywny w przypadku odnoszącym się do zbioru definicji funkcji przejścia.

Lemat 2

Funkcja „Sit" , która dostarcza kod sytuacji maszyny do momentu „t" na podstawie początkowej konfiguracji danych jest rekurencyjnie prymitywna.

Dowód:

Funkcja „Sit" jest zdefiniowana przez rekurencję. Jej wartość do momentu t = 0 jest uzyskiwana w sposób rekurencyjnie prymitywny począwszy od konfiguracji początkowej. Przejście od momentu „t" do momentu „t+1" jest realizowane przy pomocy funkcji g.

W dalszej części identyfikujemy kod związany z $Sit(t,x_1, x_2, ..., x_p)$ oraz czteroelementową sekwencję $(e, k_1, k_2, C(t))$. W szczególności $\pi_4^1(Sit(t,x_1, x_2, ..., x_p))$ jest stanem „e". Demonstracja powyższej własności jest stosunkowo prosta.

Dowód:

Czas obliczania wartości $f(x_1, x_2, ..., x_p)$ jest zadany przez:

$$T(x_1, x_2, ..., x_p) = \mu t (\pi_4^1(Sit(t,x_1, x_2, ..., x_p)) = 1),$$

gdyż maszyna osiąga tylko jeden stan końcowy q_1.

Znając sytuację do momentu $T(x_1, x_2, ..., x_p)$ możliwe jest zliczenie ilości wystąpień symbolu 1 na drugiej taśmie, która jest równa wartości $f(x_1, x_2,..., x_p)$:

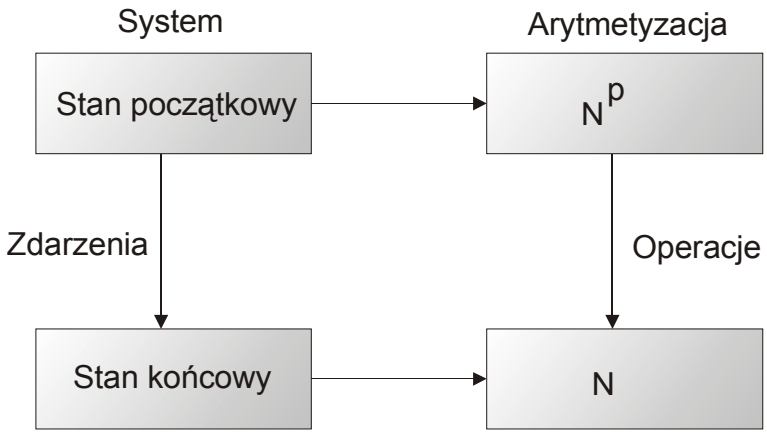
$$f(x_1, x_2,..., x_p) = \mu y(r(q(\pi_4^4(Sit(T(x_1, x_2, ..., x_p), x_1, ..., x_p)), 3^{2y+1}), 3) = 0)$$

Ta ostatnia funkcja oblicza przy pomocy funkcji q (dzielnika) i r (reszty) ilość „1" na drugiej taśmie.

8.3 Rachunek λ

Jakakolwiek byłaby dziedzina aplikacji, użytkowanie komputera sprowadza się za każdym razem do obliczania wartości funkcji. W rzeczywistości dane na wejściu są kodowane do postaci ciągu bitów interpretowanych jako sekwencje wejściowe, a na wyjściu rezultaty ponownie są kodowane i interpretowane. Etapy pośrednie realizują wszystkie istotne obliczenia.

Taka interpretacja oznacza proces arytmetyzacji, który jest szeroko używany w realizacji procesów modelowania i symulacji. Przepływ danych ilustruje rysunek nr 17.



Rysunek 17 Proces arytmetyzacji

Powyższy schemat pokazuje wagę jaką kładzie się na analizowanie w informatyce funkcji, w szczególności z przestrzeni N^p w N. Rozpatrując te ostatnie, matematycy i programiści wiedzą, że mogą one być stworzone przy wykorzystaniu zaledwie kilku funkcji prymitywnych. Przyjrzymy się teraz niektórym procedurom konstrukcji, które pomogą nam zrozumieć matematykę algorytmiczną.

Przypomnijmy, że funkcja jest regułą wiążącą obiekty, która pozwala określić wartość dla każdego zadanego jej argumentu dziedziny. Użyteczne jest zdefiniowanie tej reguły przez wyrażenie zależności pomiędzy argumentem i jego wartością. Rozumiemy przez to, np. przypisanie $x \rightarrow x^2$ lub f: $x \rightarrow f(x)$, gdzie f jest właściwym oznaczeniem reguły.

Łącząc ze sobą te dwa rodzaje zapisu dochodzimy do wyrażeń typu: „dana jest funkcja $f(x) = x^2$" , w których dostrzegamy dwuznaczność. Nawet jeżeli potrafimy dokładnie rozróżnić funkcję od jej wartości, to jak powinno się liczyć wartość wyrażenia F(f(x+1)) ? Czy należałoby liczyć na początku g(x)=f(x+1) , a następnie F(g(x)) ? Czy też może h(x) = F(f(x)) na początku, a później h(x+1) ? Wystarczy jeszcze podać operator pochodnej $D(x^2) = 2x$, by przekonać się jakie niezrozumiałości mogą zaistnieć podczas działania algorytmu.

By wyeliminować wszystkie niezrozumiałości należy zdefiniować koncepcję funkcji biorąc pod uwagę:

- samą funkcję : jako obiekt
- obiekty, do których funkcja się odnosi
- uzyskane wartości

To rozróżnienie nosi nazwę abstrakcji funkcjonalnej. Przed zapisaniem mechanizmu tego rozróżnienia, który będzie bazował również na klasycznej notacji funkcyjnej, ustalmy podstawowe zasady.

Załóżmy, że dana jest zmienna x, która może (ale nie musi) wystąpić w obiekcie E nazywanym wyrażeniem. Niech λ będzie symbolem wyróżnionym. Rozpatrzmy obiekt (λx. E) : jest on funkcją. Gdy do tego obiektu wstawiamy wartość „a" otrzymujemy wyrażenie (λx. E)a . Wartość (λx. E)a obliczamy podstawiając w wyrażeniu „E" za „x" wartość „a".

W praktyce, funkcja f notowana jako f: x $\rightarrow$ E(x) będzie się nazywała (λx.E(x)) co jest λ notacją f.

Komentarz:

1. Każde wyrażenie typu 2*5 jest liczbą, „A" jest znakiem, natomiast (λx.E) jest funkcją
2. Można interpretować obiekt (λx. E) jako rezultat zastosowania reguły oznaczanej jako λ do pary (x,E). Z tego powodu, symbol λ jest nazywany konstruktorem funkcjonalnym. Pomimo tej interpretacji należy pamiętać, że jako notacja funkcyjna, (λx. E) formuje obiekt nierozdzielny.
3. Można sobie wyobrazić, że istnieje reguła oznaczana przez @, która dla każdej pary ((λx. E),a) tworzy odpowiednio (λx. E)a , której wartość jest rezultatem podstawienia w wyrażeniu E za „x" elementu „a". Ściślej mówiąc rozpatrzmy zbiór funkcji częściowych X w Y oznaczony przez (X$\rightarrow$Y). Reguła @ może być interpretowana jako funkcja częściowa ze zbioru (X$\rightarrow$Y) $\times$ X na zbiór Y, która z każdą parą (f,x) utożsamia f(x). Z tego powodu reguła @ jest naturalnie nazywana aplikacją. Korzystnie jest zamienić @ przez konkatenację pisząc fx zamiast @(f,x) . To wyjaśnia dlaczego używa się niekiedy notacji fx dla wyrażenia wartości funkcji f na x.
4. Podstawienie wymaga prawidłowego sformalizowania. Zmienna x służy tylko do wskazania pozycji, gdzie ma miejsce podstawienie. Sama nazwa zmiennej x nie gra roli. W konsekwencji możemy, co jest bardzo wskazane, zmienić nazwę zmiennej, do której podstawiamy wartość, jeżeli może wystąpić błędne zrozumienie formuły. Opisaną sytuację przedstawiają dwa przykłady:

- dwie funkcje (λx. (2*x+1)) i (λt. (2*t+1)) są równe
- nie zapisuje się (λx. (2*x+1))x , lecz (λt. (2*t+1))x .

Przykłady:

1. Rozpatrzmy teraz wyrażenie x^2+x+1. Jeśli rozpatrzymy x jako zmienną, wtedy (λx. (x^2+x+1)) jest funkcją oznaczaną zazwyczaj jako f:x$\rightarrow x^2+x+1$; odwrotnie: λ - notacją funkcji g: x$\rightarrow$2x+3 jest (λx. 2x+3)).
2. Weźmy wyrażenie 3x-xy+5. Jeśli chcemy zrobić z 3x-xy+5 funkcję (częściową) zależną od y (dla ustalonego y) , wtedy zapisujemy (λy. (3x-xy+5)) co odpowiada notacji klasycznej f_x: y$\rightarrow$3x-xy+5. Począwszy od (λy. (3x-xy+5)) napiszmy (λx. (λy. (3x-xy+5))), co odpowiada zapisowi funkcji 2 zmiennych f: (x,y) $\rightarrow$ 3x-xy+5; odwrotnie możemy interpretować wyrażenie (λy. (λx. (3x-xy+5))) jako λ - notację funkcji g: (y,x) $\rightarrow$ 3x-xy+5 . Zaczęliśmy właśnie rozważania funkcji wielu zmiennych. W tym przypadku λ - notacja funkcji f: (x_1, x_2, ..., x_n) $\rightarrow$ E(x_1, x_2, ..., x_n) będzie przyjmowała postać: λx_1. (λx_2. (... (λx_n. E(x_1, x_2, ..., x_n)) ...)
3. Zanotujmy operator wyrażający pochodną: D(λx. x^2) = (λx. 2x)
4. Niech E będzie wyrażeniem zawierającym x, a T operatorem zdefiniowanym jako T(λx. E(x)) = (λx. E(x+1)) . Weźmy funkcję F; dwiema możliwymi interpretacjami F(f (x + 1)) będą: F(T(λx. f(x))) oraz T(F(λx. f(x))) .
5. Kiedy reguła aplikacji zapisuje się przez konkatenację, dobrze jest wyróżnić multiplikację. Przykładowo:

C::= (λx. (x^2+2*x+1))
S::= (λx. (λy. (x+y)))
P::= (λx. (λy. (x*y)))
Mamy odpowiednio C3 = 3^2+2*3 +1 = 16
S2 = (λx. (λy. (x+y)))2 = (λy. (2+y)); (S2)5 = (λt. (2+t))5 = 7
Jeśli f jest funkcją jednej zmiennej i ma jeden argument, wtedy:
P(Fa) = (λy. (Fa*y)),
W szczególności: P(C3) = (λy. (16*y)) ; P((S1)x) = (λy. ((1+x) *y))

Język λ term

Formalizm przeznaczony do opisania mechanizmów przedstawionych powyżej został wprowadzony w pracach A. Churcha, który jako pierwszy użył języka formalnego nazwanego λ-calcul. Język ten jest używany wtedy, gdy chcemy opisywać funkcje, stąd Church użył greckiej litery λ ,która symbolizuje rzymską literę L, by w ten sposób zasugerować, że jego system formalny jest językiem.

Jak to zwykle bywa, dany jest nieuporządkowany zbiór Var symboli zwanych zmiennymi oraz cztery symbole specjalne: „ λ ” „.” „(” „)” . Język λ - term jest generowany przez gramatykę:

<lambda-term> → <Var> | <lambda-term><lambda-term>
| λ <Var> . <lambda-term>

Pierwsza reguła ustala zbiór *atomów*. Dwie następne reguły formalizują odpowiednio *aplikację* i *abstrakcję*. Można stosunkowo łatwo pokazać, że powyższa gramatyka nie jest dwuznaczna.

Komentarz:

1. Każdy λ - term reprezentuje funkcję jednej zmiennej, której argumenty i wartości mogą same być funkcjami.
2. Term oznaczony jako FX jest nazywany *aplikacją* (lub *a-wyrażeniem*). Reprezentuje on połączenie operatora F z operandem X.
3. Term postaci $(\lambda x_1. (\lambda x_2. (... (\lambda x_n. T) ...)))$, gdzie $(x_1, x_2, ..., x_n)$ jest listą zmiennych, a T jest termem nazywamy abstrakcją (lub *λ-wyrażeniem*). Reprezentuje on funkcję zmiennej $X=(x_1, x_2, ..., x_n)$. Mówi się, że X jest listą *parametrów*, a T *ciałem* abstrakcji. Uzgodnimy ponadto, że term $(\lambda x_1. (\lambda x_2. (... (\lambda x_n. T) ...)))$ będzie zapisywany jako $\lambda x_1 \lambda x_2 \lambda x_n.T$.
4. W celu uproszczenia zapisu, zaadoptujemy dwie inne konwencje:
   - dla a-wyrażeń, nawiasowanie będzie występować tylko od lewej do prawej
   - dla λ-wyrażeń, ich ciało będzie się rozszerzać możliwie jak najdalej w prawą stronę

Przykład:

Załóżmy, że mamy dwie zmienne: x i y. Dla przedstawionych term

x; y; (xy); (λx. (xy)); (y(λx. (xy)); (λx. (λy. (y(λx. (xy)))))

możemy zastosować uproszczony zapis, otrzymując:

xy; λx. xy; y(λx.xy); λxy. y(λx. xy)

Obliczenia na λ termach

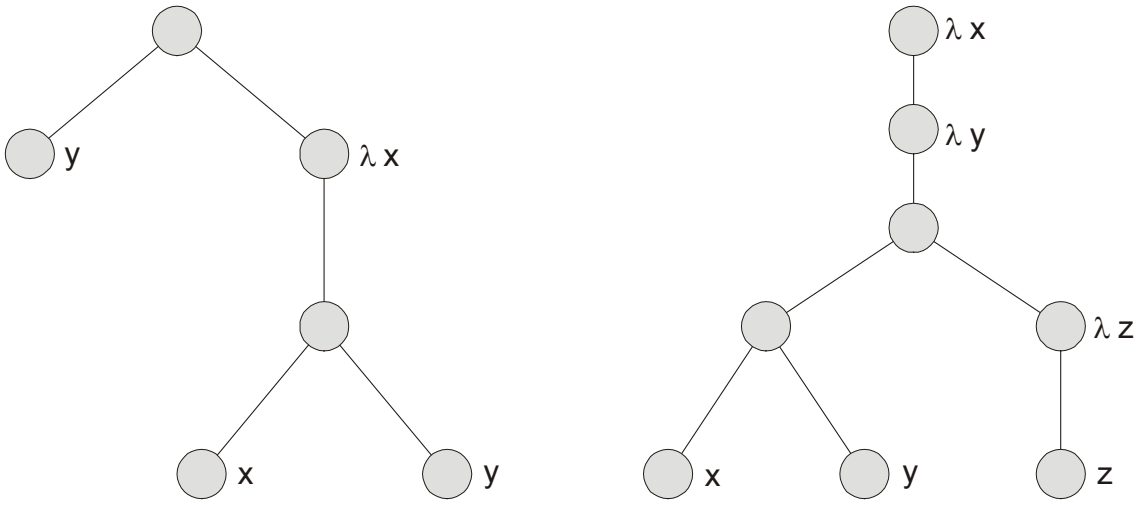
Formalnie, każda forma jest tworzona dzięki dwóm regułom, którymi są abstrakcja i aplikacja. Każdą formę możemy reprezentować przez drzewo binarne i w zależności od stopnia $d^+(s)$ każdego wierzchołka s, wyróżniamy następujące przypadki:

- $d^+(s) = 0$: s jest zmienną

- $d^+(s) = 1$: s reprezentuje abstrakcję
- $d^+(s) = 2$: s reprezentuje aplikację.

Przykład:

Respektując powyższe przypadki, termy: y(λx. xy) oraz λxy. xy(λz. y) są reprezentowane przez następujące drzewa:



Rysunek 18 Drzewa reprezentujące termy

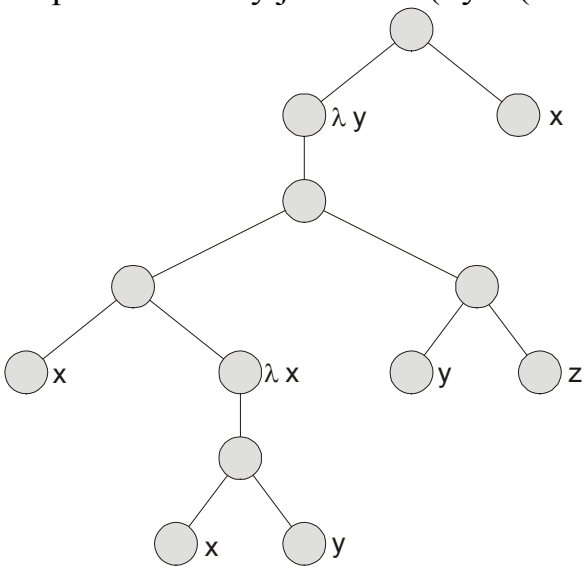
Począwszy od definicji termy , można zdefiniować indukcyjnie zbiór podzbiorów termy.

Graficznie pod – termy danej termy T są reprezentowane przez poddrzewa drzewa syntaktycznego T. Zmienna x, która figuruje na liście parametrów abstrakcji i znajduje się pod – termem T jest nazywana *połączoną*. *Wystąpienie* x jest połączone, jeśli znajduje się ono w ciele abstrakcji, której lista parametrów zawiera x. Wystąpienie zmiennej jest nazywane *wolnym*, jeśli nie jest ono połączone. Zmienna jest wolna, jeśli pozwala ona na wolne wystąpienie.

Zmienna połączona gra rolę parametru formalnego, który służy do wskazania miejsca w ciele abstrakcji, stąd jego nazwa nie grali. Ponadto dozwolone jest zmienianie nazwy zmiennej połączonej. Mówiąc językiem formalnym: niech x będzie zmienną połączoną formy T. Istnieje pod-drzewo minimalne (w sensie inkluzji) posiadające jako korzeń λx, które reprezentuje term postaci λx. M . Niech y będzie zmienną nie występującą w M. Zamiana x na y w λx. M pociąga za sobą podmianę wszystkich x na y. W ten sposób można zawsze podejrzewać, że w każdej termie, co więcej, w zbiorze term, nazwy zmiennych połączonych są różne od zmiennych wolnych.

Przykład:

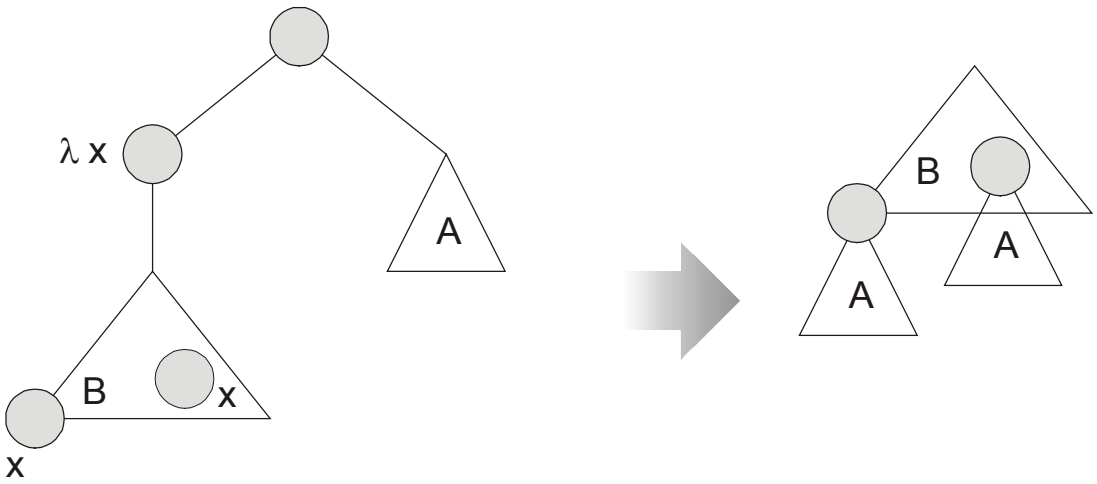
Na rysunku nr 19 przedstawiony jest term: (λy. x(λx. xy)(yz))x



Rysunek 19 Zmienne wolne i połączone

Zmienne x i y są połączone; x i z posiadają wystąpienia wolne. Aby zapobiec kolizji, wystarczy zmienić nazwę zmiennej połączonej x. Otrzymujemy wtedy (λy. x(λt. ty)(yz))x , co po zamianie y na u można zapisać jako: (λu. x(λt. tu)(uz))x.

Jeśli B jest termem zawierającym zmienną wolną x, wtedy λx. B reprezentuje funkcję, której wartość dla A, (λx. B)A otrzymuje się przez podstawienie A do wszystkich wystąpień x w B; graficznie - podłącza się korzeń drzewa reprezentującego A do wszystkich liści etykietowanych przez x w drzewie reprezentującym B, co przedstawia rysunek nr 20.



Rysunek 20 Zmiana nazw zmiennych w drzewie

Dokonana zamiana nazywa się skurczeniem wyrażenia (λx. B)A . W rezultacie (λx. xy)A kurczy się do Ay. Skurczenie może być interpretowane jako reguła ponownego przepisania. Sekwencja skurczeń nazywa się redukcją. Skurczenie oznacza się symbolem →.

Obliczenia na λ termach składają się więc z sekwencji podmian zmiennych połączonych oraz skurczeń.

Przykład:

Obliczmy (λy.x((λx.xy)(xyz)))(xy). Dla uniknięcia wszelkich nieporozumień zamieńmy nazwy zmiennych połączonych x (na u) oraz y (na v). Otrzymujemy (λv. x((λu. uv)(xvz)))(xy) . Pod – term (λu. uv)(xvz) redukuje się do (xvz)v. Term początkowy postaci (λv. x((xvz)v))(xy) redukuje się więc do x((x(xy)z)(xy)).

8.4 Klasy złożoności

Różnego rodzaju algorytmy potrzebują do działania różnych czasów obliczeń. Zarówno ilość jednostek obliczeniowych, jak i rodzaj danych wejściowych mają na ten czas duży wpływ, więc do określenia efektywności (i zarazem prędkości) algorytmów trzeba używać innych wielkości.

Prędkość algorytmów przedstawia się przede wszystkim jako funkcję zależną od danych wejściowych. Jeśli np. wykonanie inkrementacji zmiennej całkowitej wymaga czasu „k", to złożoność obliczeniowa algorytmu, który inkrementuje N liczb wynosi N*k.

```
For i:=1 to i=N do
        Inc(liczba[i])
```

Ostatecznie czas wykonania algorytmu jest funkcją zależną od wszystkich danych wejściowych algorytmu, np. $f(x_1, x_2, x_3) = 3*x_1^2 - x_2 + 2*x_3$.

Mnogość funkcji powyższego typu i trudności w ich porównywaniu stały się powodem wprowadzenia tzw. klas złożoności obliczeniowej. Zostały wprowadzone notacje Θ(n), O(n) oraz Ω(n), które tworzą odpowiednie *klasy równoważności*.

Notacja Θ(n)

Dla danej funkcji g(n) Symbol Θ(g(n)) oznacza zbiór funkcji

$\Theta(g(n)) = \{f(n)$: istnieją stałe c_1, c_2, n_0 takie, że dla wszystkich $n \geq n_0$ zachodzi:

$0 \leq c_1*g(n) \leq f(n) \leq c_2*g(n)$ }

Innymi słowy, funkcja f(n) jest klasy $\Theta(g(n))$, jeśli od pewnego n może być ona ograniczona z dołu i z góry przez funkcje będące wielokrotnościami funkcji g(n). Przykładowo, możemy łatwo pokazać, że funkcja $f(n) = 3n^2 + n \in \Theta(n^2)$. W tym celu należy tak dobrać parametry c_1, c_2, by od pewnego n zachodziło:

$c_1*n^2 \leq 3n^2 + n \leq c_2*n^2$ po podzieleniu przez n^2 otrzymujemy

$c_1 \leq 3 + 1/n \leq c_2$

Łatwo spostrzec, że gdy przyjmiemy wartości: $c_1 = 2$, $c_2 = 4$, to dla dowolnego $n > 0$ powyższa nierówność będzie spełniona. W tym miejscu należy dodać, że mimo istnienia wielu dopuszczalnych par wartości c_1, c_2, sprawdzenie, czy funkcja jest klasy $\Theta(n^2)$ sprowadza się do znalezienia zaledwie jednej takiej pary.

Notacja O(n)

Dla danej funkcji g(n) Symbol O(g(n)) oznacza zbiór funkcji

$O(g(n)) = \{f(n)$: istnieją stałe c_2, n_0 takie, że dla wszystkich $n \geq n_0$ zachodzi:

$0 \leq f(n) \leq c_2*g(n)\ \}$

Innymi słowy, funkcja f(n) jest klasy O(g(n)), jeśli od pewnego n może być ona ograniczona od góry przez funkcję będącą wielokrotnością funkcji g(n).

Notacja Ω(n)

Dla danej funkcji g(n) Symbol Ω (g(n)) oznacza zbiór funkcji

$\Omega(g(n)) = \{f(n)$: istnieją stałe c_1, n_0 takie, że dla wszystkich $n \geq n_0$ zachodzi:

$0 \leq c_1*g(n) \leq f(n)\ \}$

Innymi słowy, funkcja f(n) jest klasy $\Omega(g(n))$, jeśli od pewnego n może być ona ograniczona od dołu przez funkcję będącą wielokrotnością funkcji g(n).

Z powyższych definicji notacji $\Theta(n)$, $O(n)$ oraz $\Omega(n)$ wynika, że dla dowolnych dwóch funkcji f(n) oraz g(n) zachodzi:

$f(n) \in \Theta(g(n))$ *wtedy i tylko wtedy, gdy* $f(n) \in O(g(n))$ i $f(n) \in \Omega(g(n))$

8.5 NP-zupełność

Klasa języków P

Jeśli dla deterministycznej maszyny Turinga M istnieje taki wielomian p(n), taki że M nigdy nie wykonuje więcej niż p(n) ruchów przy rozpoznawaniu słowa długości n, wówczas o M mówi się, że jest *maszyną czasu wielomianowego*. Na klasę języków P składają się języki akceptowane przez deterministyczne maszyny Turinga czasu wielomianowego. Alternatywnie, P składa się z problemów, które mogą zostać rozwiązane na prawdziwym komputerze przy pomocy algorytmu działającego w czasie wielomianowym.

Wiele znajomych problemów jest klasy P np. problem sortowania szybkiego, problem znajdowanie minimalnego drzewa rozpinającego, problem znajdowania minimalnej ścieżki między dwoma dowolnymi wierzchołkami grafu itp. Wszystkie problemy P są więc rozwiązywalne przez algorytmy wielomianowe.

Klasa języków NP

Niedeterministyczna maszyna Turinga, która nigdy nie wykonuje więcej niż p(n) ruchów (dla jakiegoś wielomianu p) przy dowolnym wyborze przejść nazywana jest *niedeterministyczną maszyną Turinga czasu wielomianowego*. Na klasę języków NP składają się języki akceptowane przez niedeterministyczne maszyny Turinga czasu wielomianowego. Alternatywnie, P składa się z problemów, które mogą zostać rozwiązane na prawdziwym komputerze przy pomocy algorytmu działającego w czasie nie-wielomianowym.

Wiele problemów, które należą do klasy NP. wydaje się nie należeć do klasy P. Przykładowo:

- Problem komiwojażera (czy istnieje w grafie ścieżka długości mniejszej od k obchodząca wszystkie wierzchołki); Dla grafu o n wierzchołkach na problem ten jest rozwiązywalny algorytmem o złożoności n! (sprawdzenie, czy dana ścieżka jest rozwiązaniem problemu zajmuje jednak czas O(n)).
- Problem SAT (sprawdzenie, czy formuła logiczna może być spełniona, czyli czy istnieje takie wartościowanie zdań reguły, dla której jej wynik jest prawdziwy). Dla reguły składającej się z n zdań logicznych problem ten jest rozwiązywalny algorytmem o złożoności 2^n (sprawdzenie czy dane wartościowanie rozwiązuje problem zajmuje jednak czas O(n)).
- Problem CLIQUE (sprawdzenie, czy graf posiada podzbiór k wierzchołków połączonych krawędziami „każdy z każdym”). Dla rozwiązania tego problemu nie znaleziono dotychczas algorytmu działającego w czasie wielomianowym. Jeśli jednak mamy

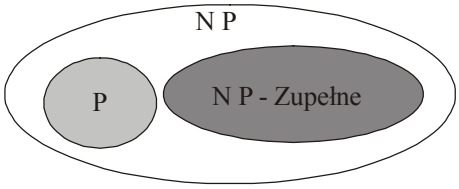
przypuszczalnie dobre rozwiązanie, to możemy je sprawdzić w czasie $O(k^2)$.

Jedną z największych zagadek naszych czasów jest pytania, czy istnieje jakiś problem w NP, który nie jest w P.

Problemy NP zupełne

Jeśli nie jesteśmy w stanie odpowiedzieć na pytanie, czy P = NP, możemy przynajmniej pokazać, że niektóre problemy NP są *najtrudniejsze* tzn. jeśli jakikolwiek z nich byłby w P, wówczas P = NP. Formalnie zbiór najtrudniejszych problemów NP nosi nazwę *NP-zupełny*. Każdy nowy, trudny do rozwiązania problem, dla którego udowodnimy, że jest NP-zupełny utwierdza badaczy naukowych w przekonaniu, iż wszystkie problemy NP-zupełne są trudne do rozwiązania.

Wydaje się, że wzajemne relacje inkluzji problemów spełniają schemat pokazany na rysunku nr 21.



Rysunek 21 Klasyfikacja problemów o różnej złożoności obliczeniowej

Udowadnianie NP-zupełności problemów

Jeśli dowolna instancja problemu P_1 może zostać przekształcona do instancji problemu P_2 przy pomocy algorytmu działającego w czasie wielomianowym (ze względu na rozmiar danych wejściowych), to mówimy, że P_1 jest *redukowalny w czasie wielomianowym* (*RCZW*) do P_2.

Łatwo wykazać, że jeśli P_1 RCZW P_2, oraz P_2 jest w P, to również P_1 jest w P. Dowód opiera się na pokazaniu, że jeśli P_2 jest rozwiązywany w czasie $O(w_2(n))$, a RCZW dokonywana jest w czasie $O(r(n))$, to problem P_1 jest rozwiązywany w czasie $O(r(n)*w_2(n))$. Iloczyn dwóch wielomianów jest wielomianem, stąd P_1 jest rozwiązywany w czasie wielomianowym, czyli należy do P.

Udowadnianie NP-zupełności nowych problemów powinno zostać poprzedzone udowodnieniem, że *każdy* problem NP-zupełny jest RCZW do problemu SAT. Jako, że dowody takich redukcji dla znanych problemów NP-zupełnych są obszerne, nie będą one zamieszczone w tym rozdziale. Nie pokażemy również dowodu bardziej ogólnego twierdzenia:

Każdy problem L ze zbioru NP jest RCZW do problemu SAT.

Jeśli wychodzimy z założenia, że *każdy* problem NP-zupełny jest RCZW do dowolnego innego problemu NP-zupełnego, to by udowodnić, że nowy problem jest NP-zupełny należy najpierw dowieść, że jest on NP, a następnie pokazać, że jest on RCZW do jakiegoś znanego problemu NP-zupełnego.

9. Klasy algorytmów

Streszczenie: *W rozdziale tym pokazane zostaną podstawowe techniki stosowane przy projektowaniu algorytmów, w szczególności: Branch & Bound, programowanie dynamiczne, dziel i zwyciężaj, algorytmy zachłanne oraz algorytmy z nawrotami. Po formalnym omówieniu każdej techniki przedstawimy przykład jej zastosowania wraz z implementacją w wybranym języku programowania.*

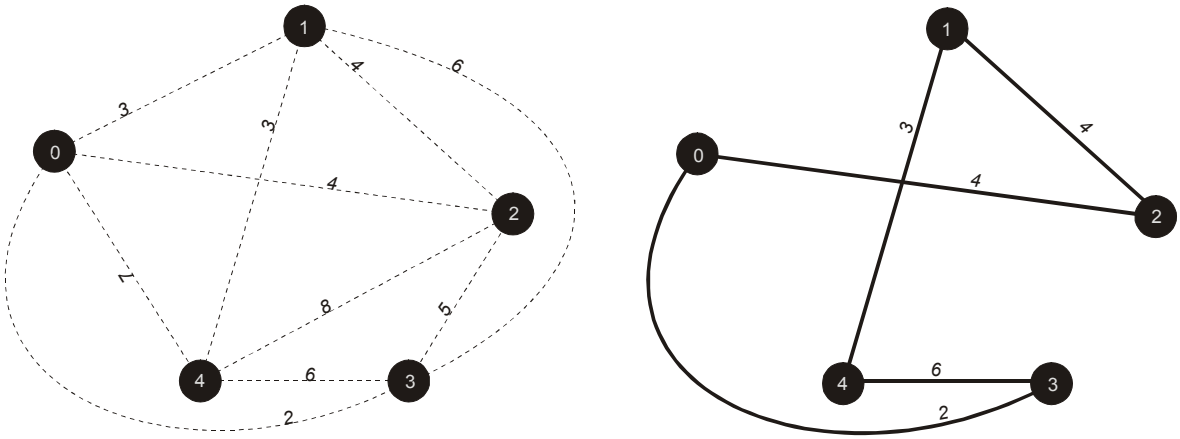
9.1 Branch & Bound

Metoda ta znajduje szczególne zastosowanie przy problemach, które wymagają dokładnego rozwiązania. Często analiza problemu zmusza nas do poszukiwania najlepszego rozwiązania, stąd zachodzi potrzeba rozpatrzenia wszystkich możliwości. Sprawne przeglądanie wszystkich możliwości przypomina niekiedy budowanie drzewa możliwych rozwiązań. Skomplikowane drzewa są nieraz tak duże, że analiza wszystkich ich gałęzi jest bardzo żmudna.

W tym momencie możemy wykorzystać metodę Branch & Bound, czyli obcinanie gałęzi drzewa, które nie prowadzą do lepszego rozwiązania niż aktualnie osiągnięte. Gałęzie są obcinane również wówczas, gdy prowadzą do nieprawidłowych rozwiązań.

Mówiąc o porównywaniu rozwiązań należy zwrócić uwagę na konieczność funkcji szacującej dane rozwiązanie, bądź też analizującej rozwiązanie w trakcie jego powstawania.

By dokładniej zrozumieć zagadnienie obcinania gałęzi drzewa rozpatrzymy klasyczny problem komiwojażera. Zakładamy, że komiwojażer startuje z węzła „0” i próbuje odwiedzić wszystkie węzły grafu nie-skierowanego, tak jak na rysunku nr 1. Dodatkowym ograniczeniem będzie, że może on tylko raz przejść daną krawędzią. Z każdą krawędzią związany jest koszt, a optymalne rozwiązanie polega na odwiedzeniu wszystkich wierzchołków po ścieżce o najmniejszym koszcie.



Rysunek 22 Graf połączeń dla komiwojażera i najkrótsza marszruta

Na podstawie powyższego rysunku łatwo narysować drzewo, które obrazuje wszystkie możliwości odwiedzenia kolejnych wierzchołków. Poniżej pokazany jest szkic takiego drzewa, mającego jako korzeń węzeł „0".

0																											
1						2						3						4									
2		3		4		1		3		4		1		2		4		1		2		3					
3	4	2	4	2	3	3	4	1	4	1	3	2	4	1	4	1	2	2	3	1	3	1	2				
4	3	4	2	3	2	4	3	4	1	3	1	4	3	4	1	2	1	3	2	3	1	2	1				
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0				

Przejście każdą gałęzią drzewa do liścia (węzła zawierającego ponownie „0") gwarantuje znalezienie rozwiązania, jednak tylko przyciemnione węzły znajdują się na dwóch najlepszych ścieżkach. Zaciemnione węzły reprezentują rozwiązanie pokazane na drugim grafie z rysunku nr 1.

Pierwszym sposobem na usprawnienie procesu poszukiwania najlepszego rozwiązania może być obcinanie gałęzi powyższego drzewa, które prowadzą do rozwiązania gorszego niż aktualnie osiągnięte. Przeanalizujmy proces poszukiwania rozwiązania z podanego przykładu. Załóżmy, że algorytm znalazł już pierwszą optymalną ścieżkę: 0→2→1→4→3→0. Koszt przejścia tej ścieżki, to 19. Gdy algorytm będzie później podążał ścieżką 0→4→2, wówczas już na tym etapie będzie widział, że dodanie kolejnego wierzchołka do ścieżki zwiększy jej koszt na tyle, że nie będzie ona lepsza od ścieżki 0→2→1→4→3→0. Nie ma więc potrzeby dalszego rozpatrywania węzłów poddrzewa, którego korzeń jest wskazywany przez ścieżkę 0→4→2. Wszystkie gałęzie tego poddrzewa są zatem obcinane. Zysk obcinania gałęzi byłby jeszcze większy, gdyby np. krawędź 0→4 miała koszt > 19, gdyż wtedy obcięlibyśmy ¼ drzewa poszukiwań.

Funkcja szacująca powstające rozwiązanie sumuje koszt przejścia po krawędziach, które już leżą na ścieżce i jeśli ten koszt okaże się większy od najlepszego rozwiązania, gałąź zostaje obcięta.

W dalszej części przyjrzymy się innemu algorytmowi poszukiwania ścieżki optymalnej. Zamiast próbować dobierać kolejne wierzchołki ścieżki na kolejnych poziomach drzewa, będzie on starał się na każdym poziomie drzewa klasyfikować daną krawędź jako leżącą na optymalnej ścieżce, lub nie leżącą na optymalnej ścieżce. Zagłębianie się będzie polegało na zamianie krawędzi „możliwych" na „niemożliwe" lub „pewne". Jeśli graf wyjściowy nie będzie już posiadał krawędzi „możliwych", wówczas zostanie osiągnięty liść w drzewie poszukiwań. Na etapie klasyfikowania krawędzi jako „pewna" niektóre krawędzie będą automatycznie usuwane zmniejszając rozmiar drzewa poszukiwań. Np. jeśli krawędź (2,1) oraz (1,4) zostaną potraktowane jako „pewne", wówczas wszystkie inne krawędzie wchodzące do węzła „1" muszą zostać sklasyfikowane jako „niemożliwe".

Optymalne rozwiązanie może zostać znalezione po usunięciu 5 krawędzi: (0,3), (0,1), (0,4), (4,2), (0,2).

Drugi algorytm posiada trudniejszą funkcję szacującą. Liczy ona minimalny koszt ścieżki na danym etapie przez włączenie do niej w pierwszej kolejności krawędzi „pewnych", a następnie minimalnych krawędzi „możliwych" wychodzących z kolejnych wierzchołków.

W przedstawionym dalej algorytmie tłustym drukiem zostały wyróżnione miejsca: (1) porównanie aktualnego rozwiązania z najlepszym, (2) zamiana krawędzi „możliwej" na „pewną" lub „niemożliwą", (3) automatyczne usuwanie krawędzi, (4) funkcja szacująca.

Na początku zdefiniujemy jak w poniższym algorytmie wygląda graf:

```
struct Edge{                    //Struktura związana z krawędzią
        int cost;               //      koszt przejścia krawędzi
        int color;              //     -1 krawędź niemożliwa
};                              //      0 krawędź możliwa
                                //      1 krawędź pewna
Edge[vCount][vCount] G;         //Graf wszystkich możliwych dróg voyagera
Edge[vCount][vCount] GB;        //Najlepszy graf drogi TSP
```

Następnie przedstawimy zmienne globalne i procedury: Dla uproszczenia zapisu nie będziemy w nagłówkach procedur pisać typu grafu „Edge[vCount][vCount]".

```
int BestCost = MaxInt;
int SureEdge=0;                          //ilość krawędzi "pewnych" w ścieżce voyagera
int vCount ;                             //ilość wierzchołków w grafie G
int N_COUNT(Gtemp,int a, int eClass);    //Ilość sąsiadów typu eClass dla wierzchołka a
                                         //w grafie Gtemp
int FIRST(Gtemp,int a);                  //Zwraca pierwszego sąsiada wierzchołka a w
                                         //grafie Gtemp
int NEXT(Gtemp,int a, int b);            //Zwraca kolejnego (po b) sąsiada wierzchołka a
                                         //w grafie Gtemp
PATH(Gtemp);                             //Rekurencyjna procedura poszukiwania
                                         //optymalnej ścieżki w grafie Gtemp
GET_GRAPH(G);                            //Pobranie struktury grafu G
COUNT_MIN_COST(Gtemp);                   //Wyliczenie minimalnego kosztu przebycia
                                         //ścieżki voyagera dla grafu Gtemp
```

Teraz napiszemy rekurencyjną procedurę PATH(Gtemp), która przy każdym wywołaniu szacuje minimalny koszt ścieżki komiwojażera dla grafu Gtemp, a następnie próbuje tą ścieżkę optymalizować przez sklasyfikowanie jednej z krawędzi „możliwych” jako krawędź „niemożliwa” lub „pewna”. Procedura PATH wywołuje funkcję szacującą minimalny koszt COUNT_MIN_COST(Gtemp).

```
PATH(Gtemp){
        int a, b, i, result, min_cost;
        min_cost = COUNT_MIN_COST(Gtemp)

        // Jeżeli możliwe jest polepszenie ścieżki voyagera
        if(min_cost<BestCost)                                              (1)
        {
                if(SureEdge==ilosc*2)
                {       //Zostały zdefiniowane wszystkie "pewne" krawędzie
                        //Aktualizacja najlepszego grafu przejść voyagera (GB)
                        BestCost=min_cost;
                        for(i=0;i<ilosc;i++)
                        for(j=0;j<ilosc;j++) GB[i][j].color=Gtemp[i][j].color;
                }
                else
                {       //Można wybrać jakąś krawędź "możliwą"

                        <Pętla po krawędziach znajdująca jakąś krawędź "możliwą" (a,b)>

                        //Krawędź (a,b) obcinamy, jeśli nie narusza to spójności grafu
                        //Gtemp, a następnie rekurencyjnie wywołujemy PATH(Gtemp)
                        Gtemp[a][b].color=-1;                              (2)
                        Gtemp[b][a].color=-1;
                        if(N_COUNT(Gtemp, a ,-1) < vCount-2)&&
                        if(N_COUNT(Gtemp, b ,-1) < vCount-2) PATH(Gtemp)

                        //A teraz krawędź (a,b) oznaczamy jako "pewną"
```

```
                    //Jeśli do wierzchołka a lub b dochodzą już 2 krawędzie "pewne",
                    //wówczas dochodzące do nich krawędzie "możliwe" zostają obcięte
                    //Jeśli do wierzchołka a lub b dochodzą już 3 krawędzie "pewne"
                    //wówczas nie wywołujemy rekurencyjnie PATH(Gtemp)
                    int result=1;
                    Gtemp[a][b].color=1;                                    (2)
                    Gtemp[b][a].color=1;
                    <Pętla dla wierzchołka w = 1,2>{
                            i=N_COUNT(Gtemp,w,1);
                            if(i>2) result=0;
                            if(i==2) {
                                    int k=FIRST(Gtemp,a);
                                    while(k!=-1) {
                                            if(Gtemp[a][k].color==0){
                                                    Gtemp[a][k].color=-1;   (3)
                                            //czy nie zburzymy spójności grafu przy wierzchołku k
                                                    if(N_COUNT(Gtemp,k,-1) > vCount – 3)
                                                            result=0;
                                                    }
                                                    k=NEXT(Gtemp,a,k);
                                            }
                                    }
                            }
                            if(result) PATH(Gtemp);

                    } // koniec manipulacji krawędzią "możliwą"
            } // koniec polepszania ścieżki voyagera
}
```

Dalej przedstawiona jest procedura inicjująca graf wejściowy dla algorytmu. Aktualnie jest nim graf omawiany na początku sekcji Branch & Bound.

```
GET_GRAPH(G){
        for(i=0;i<vCount;i++)
                for(j=0;j<vCount;j++){
                        G[i][j].color=-1;
                        G[j][i].color=-1; }
        G[0][1].color=0; G[1][0].color=0; G[0][1].cost=3; G[1][0].cost=3;
        G[1][2].color=0; G[2][1].color=0; G[1][2].cost=4; G[2][1].cost=4;
        G[0][2].color=0; G[2][0].color=0; G[0][2].cost=4; G[2][0].cost=4;
        G[0][4].color=0; G[4][0].color=0; G[0][4].cost=7; G[4][0].cost=7;
        G[4][2].color=0; G[2][4].color=0; G[4][2].cost=8; G[2][4].cost=8;
        G[2][3].color=0; G[3][2].color=0; G[2][3].cost=5; G[3][2].cost=5;
        G[3][4].color=0; G[4][3].color=0; G[3][4].cost=6; G[4][3].cost=6;
        G[0][3].color=0; G[3][0].color=0; G[0][3].cost=2; G[3][0].cost=2;
        G[1][4].color=0; G[4][1].color=0; G[1][4].cost=3; G[4][1].cost=3;
        G[1][3].color=0; G[3][1].color=0; G[1][3].cost=6; G[3][1].cost=6;
}
```

Procedury: NEXT, FIRST, N_COUNT zostały pominięte. Przedstawiono natomiast funkcję szacującą:

```
int COUNT_MIN_COST(Gtemp){                                                    (4)
        int min_cost=0,i;
        for(i=0;i<ilosc;i++){
                int found_edge_count=0;
                int first_found=-1;
                j=FIRST(Gtemp,i);
                while(found_edge_count<2){
                        j=FIRST(Gtemp,i);
                        int min_edge=MaxInt;
                        while(j!=-1){
                                if(Gtemp[i][j].color==1){          //krawędź "pewna"
                                        SureEdge++;
                                        found_edge_count++;
                                        min_cost+=Gtemp[i][j].cost;
                                }
                                if((Gtemp[i][j].color==0)&&(Gtemp[i][j].cost<min_edge)
                                &&(j!=first_found)){               //krawędź "możliwa"
                                        min_edge=Gtemp[i][j].cost;
                                        if(found_edge_count==0) first_found=j;
                                }
                                j=NEXT(Gtemp,i,j);
                        } //Przeglądnęliśmy wszystkich sąsiadów
                        if(found_edge_count<2){
                                found_edge_count++;
                                min_cost+=min_edge;
                        }
                }
        }
        return min_cost
}
```

9.2 Programowanie dynamiczne

Metoda programowania dynamicznego znajduje szczególne zastosowanie wszędzie tam, gdzie zachodzi konieczność poszukiwania najlepszego rozwiązania budowanego na podstawie rozwiązań mniejszych. Najczęściej proces programowania dynamicznego możemy podzielić na równe, podobnie wyglądające etapy ułożone hierarchicznie. W każdym etapie zostaje wyznaczone rozwiązanie lokalne. Etap wyższy korzysta z wyników uzyskanych w etapach niższych i zwraca wynik, który z kolej będzie wykorzystany przez etap jeszcze wyższy. Wyniki z poszczególnych etapów obliczeń muszą być zapamiętywane w celu ich późniejszego wykorzystania.

W celu zobrazowania efektywności programowania dynamicznego przeanalizujemy przykład dynamicznego mnożenia macierzy. Jak wiadomo, przy mnożeniu ciągu macierzy $A_1 * A_2 * ... * A_n * ... * A_N$ musi być spełniona zasada: ilość kolumn macierzy stojącej po lewej stronie operatora mnożenia musi odpowiadać ilości wierszy macierzy stojącej po prawej stronie tego operatora. Przestawianie macierzy (jeśli oczywiście będzie zgodne z wcześniejszą zasadą) wpływa na wynik, jednak kolejność wykonywania operacji mnożenia macierzy wyniku nie zmienia. To, czy pomnożymy najpierw $(A_1 * A_2) * A_3$, czy też $A_1 * (A_2 * A_3)$ będzie miało wpływ na czas wykonywania obliczeń, bowiem w obu przypadkach będziemy mieli do czynienia z różną ilością elementarnych operacji mnożenia.

Przeanalizujmy prosty przykład: mamy ciąg 5 macierzy: A_1, A_2, A_3, A_4, A_5, o wymiarach [1 x 4], [4 x 1], [1 x 4], [4 x 1], [1 x 4], *przy czym pierwszy wymiar jest ilością kolumn, a drugi jest ilością wierszy macierzy*. Jeśli zastosujemy kolejność mnożenia $(((A_1 * A_2) * A_3) * A_4) * A_5$, wówczas elementarnych operacji mnożenia będzie: (((4*4) + 4*4) + 4*4) +4*4 = 64. Jeśli natomiast zastosowalibyśmy kolejność mnożenia macierzy $A_1 * ((A_2 * A_3) * (A_4 * A_5))$, wtedy ilość elementarnych operacji mnożenia byłaby równa 4 + (4 + 1 + 4) = 13, czyli niemal 5 razy mniejsza niż w przypadku pierwotnym.

Nim posłużymy się programowaniem dynamicznym w celu znalezienia optymalnego nawiasowania, pokażemy jak można to zrobić stosując metodę rekurencyjną dziel i zwyciężaj, o której będzie mowa później.

Oznaczmy wymiary ciągu macierzy A_1, A_2 , ..., A_n , ... , A_N jako r_0, r_1, r_2, ..., r_n, ...r_N gdzie macierz A_1 ma wymiary: $[r_0 \text{ x } r_1]$,macierz A_n: $[r_{n-1} \text{ x } r_n]$, a macierz A_N: $[r_{N-1} \text{ x } r_N]$.

Rekurencyjny algorytm wyznaczający ilość elementarnych operacji mnożenia ma postać:

$$MinMul(Ai, Aj) = \begin{cases} 0 \quad gdy \ i = j \\ r_{i-1} * r_j \quad gdy \ i+1 = j \\ r_{i-1} * r_j + \min_{i \le k < j} (MinMul(A_i, A_k) + MinMul(A_{k+1}, A_j)) \quad gdy \ i+1 < j \end{cases}$$

Ilość elementarnych operacji mnożenia dla $(A_1 * A_2 * ... * A_n * ... * A_N)$ wynosi:

MinMul(A_1, A_N)

W celu odtworzenia zapisu nawiasowania, należałoby przechowywać w buforze wartości rozcinające „k” dla poszczególnych wywołań funkcji MinMul. Powyższy algorytm, mimo iż działa poprawnie posiada istotne wady. Oprócz tego, że jest rekurencyjny, co dla długich ciągów macierzy może wywołać problemy z pamięcią, wykonuje on wielokrotnie te same obliczenia. Przykładowo, zarówno wywołanie *MinMul(A_1, A_5),* jak i wywołanie *MinMul(A_1, A_6)* powoduje wywołanie *MinMul(A_1, A_4)*. Prowadzi to do niepotrzebnego marnowania czasu na powtarzające się obliczenia.

Stosując programowanie dynamiczne dla problemu optymalnego nawiasowania nie tylko pozbywamy się rekurencji, ale także nie wykonujemy zbędnych obliczeń.

Algorytm korzysta z macierzy buforowej, która przechowuje ilości elementarnych mnożeń dla kolejnych zakresów z wejściowego ciągu macierzy. Jeśli ilością macierzy jest N, to macierz buforująca MB ma wielkość [N x N]. Pod koniec działania algorytmu w $MB_{[i,j]}$ powinna się znaleźć ilość elementarnych operacji mnożenia koniecznych przy mnożeniu kolejnych macierzy A_i, A_{i+1}, ..., A_{j-1}, A_j. Element $MB_{[1,N]}$ będzie przechowywał liczbę reprezentującą minimalną ilość mnożeń przy liczeniu wyrażenia: $A_1 * A_2 * ... * A_n * ... * A_N$.

Algorytm działa dynamicznie, tzn. w kolejnych jego krokach liczone są odpowiednie wartości $MB_{[i,j]}$ dla coraz większego zakresu <i...j>. W zerowym kroku wykonywane jest podstawienie $MB_{[i,i]} = 0$ dla i=1..N. Szerokość zakresu zmienia się od 2 do N. W algorytmie użyjemy także macierzy CUT wielkości [N x N]. Będzie ona podawała lokalizację rozcięcia spowodowanego nawiasowaniem. Podobnie, jak macierz MB, w zerowym kroku wykonywane jest podstawienie $CUT_{[i,i]} = 0$ dla i = 1..N. Dla podanego przykładu: $A_1 * ((A_2 * A_3) * (A_4 * A_5))$ macierz MB i CUT po zakończeniu algorytmu przyjmą postać:

MB				
0				
16	0			
8	4	0		
24	8	16	0	
13	9	8	4	0

CUT				
0				
1	0			
1	2	0		
1	3	3	0	
1	3	3	4	0

Ostatecznie algorytm w składni FORTRANA 90 ma postać: (MATRIX_COUNT to ilość macierzy)

```
do i=1,MATRIX_COUNT
      MB(i,i) = 0
      CUT(i,i) = 0
end do
do l=1,MATRIX_COUNT – 1                    !l jest wielkością zakresu
      do i=1,MATRIX_COUNT - l
            j=i+l
            do k=i,j-1                     !k jest rozcięciem A_i ... A_j
               minimal=MB(i,k) + MB(k+1,j) + r(i) * r(k-1) * r(j-1)
                  if (k .eq. i) then
                        MB(i,j)=minimal
                        CUT(i,j) = k
```

```
                    ELSE if (MB(i,j) .ge. minimal) then
                        MB(i,j) = minimal
                        CUT(i,j) = k
                    end if
                end do
            end do
        end do
```

Wykorzystując macierz CUT możemy teraz optymalnie pomnożyć ze sobą macierze:

$$Mutiply(Ai, Aj) = \begin{cases} A_i \quad gdy \ i = j \\ A_i \times A_j \quad gdy \ i+1 = j \\ Multiply(A_i, A_k) \times Multiply(A_{k+1}, A_j) \quad gdy \ i+1 < j \end{cases}$$

$Mutiply(A_1, A_N) = A_1 * A_2 * ... * A_n * ... * A_N$

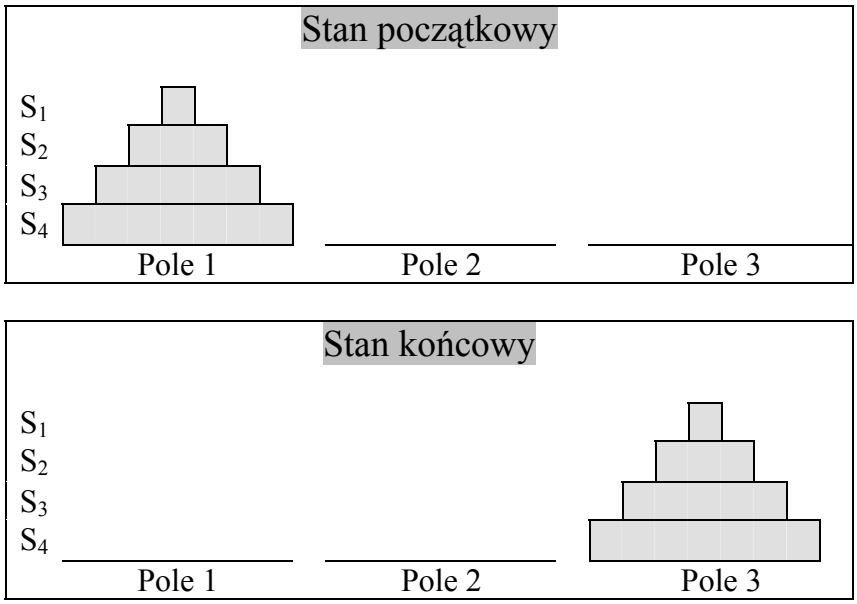
9.3 Dziel i zwyciężaj

Z metodą tą mogliśmy się już spotkać w poprzednich sekcjach. Jej głównym zadaniem jest dzielenie problemów trudnych na problemy prostsze, te z kolei na jeszcze prostsze itd. Gdy problemy są już na tyle proste, że nie wymagają dzielenia, wówczas następuje tzw. scalanie rezultatów (zwyciężaj).

Z uwagi na naturę tej metody jest ona najczęściej implementowana przy wykorzystaniu rekurencji, jednak należy pamiętać, że nie zawsze jest ona konieczna.

W poprzedniej sekcji przedstawiony zostać algorytm znajdowania optymalnego nawiasowania przy wykorzystaniu metody „dziel i zwyciężaj" jednak pokazano, iż jest on mało efektywny, stąd przedstawimy inny problem, dla którego rozwiązanie metodą „dziel i zwyciężaj" okaże się uzasadnione.

Przeanalizujemy klasyczny problem wież Hanoi. Dla przypomnienia, polega on na tym, iż są 3 pola, na których mogą być budowane wieże składające się z segmentów różnych rozmiarów. Ogólnie segmentów do budowy wież jest N: s_1, s_2, ..., s_N, przy czym każdy segment ma różną od pozostałych wielkość, więc możemy przyjąć $s_1 < s_2 < ... < s_N$, Wieża może być budowana wyłącznie przez umieszczanie na jej szczycie s_i segmentów s_j, gdzie $s_i > s_j$. W stanie początkowym istnieje tylko 1 wieża złożona z wszystkich segmentów. W stanie końcowym wieża ta ma się znaleźć w innym polu. Schemat tych dwóch stanów pokazany jest na rysunku nr 23.



Rysunek 23 Stany wież Hanoi: początkowy i końcowy

By przejść ze stanu początkowego do stanu końcowego, tak jak na rysunku nr 2 należy wykonać operacje: $S_1 \rightarrow Pole_2$, $S_2 \rightarrow Pole_3$, $S_1 \rightarrow Pole_3$, $S_3 \rightarrow Pole_2$, $S_1 \rightarrow Pole_1$, $S_2 \rightarrow Pole_2$, $S_1 \rightarrow Pole_2$, $S_4 \rightarrow Pole_3$, $S_1 \rightarrow Pole_3$, $S_2 \rightarrow Pole_1$, $S_1 \rightarrow Pole_1$, $S_3 \rightarrow Pole_3$, $S_1 \rightarrow Pole_2$, $S_2 \rightarrow Pole_3$, $S_1 \rightarrow Pole_3$.

Dla małej liczby segmentów jesteśmy w stanie „na oko" znaleźć rozwiązanie, jednak co by było, gdyby segmentów było N? Wykorzystamy tu bezpośrednio metodę dziel i zwyciężaj. Problem będziemy dzielili tak długo, aż stanie się trywialnym, czyli gdy ilością segmentów będzie 2. Tak więc 1-szy podział sprowadzi nam problem do N-1 segmentów, 2-gi do N-2 segmentów, N-2gi do 2 segmentów.

Wprowadźmy oznaczenie $<S_i..S_j>$ na grupę segmentów S_i, S_{i+1}, ..., S_{j-1}, S_j, leżących jeden na drugim. Zauważmy, że rozwiązanie dla N segmentów sprowadza się do wykonania operacji: $<S_1..S_{N-1}> \rightarrow Pole_2$, $\mathbf{S_N \rightarrow Pole_3}$, $<S_1..S_{N-1}> \rightarrow Pole_3$. Z kolei: $<S_1..S_{N-1}> \rightarrow Pole_2 = <S_1..S_{N-2}> \rightarrow Pole_3$, $\mathbf{S_{N-1} \rightarrow Pole_2}$, $<S_1..S_{N-2}> \rightarrow Pole_2$ itd.

Problem wieży N segmentowej rozdzielamy więc na problemy wież 2 segmentowych, a następnie je scalamy wykonując po kolei operacje elementarne (oznaczone tłustym drukiem). Pozostaje jeszcze problem, o którym nie było dotychczas mowy. Pierwszą operacją na grupie segmentów jest $<S_1..S_{N-1}> \rightarrow$Pole2, przy czym wiemy, że segment $<S_1..S_{N-1}>$ jest w $Polu_1$. Następne operacje na segmentach są uwarunkowane wcześniejszymi, tzn. jeżeli mamy w danym kroku wykonać $<S_1..S_j> \rightarrow Pole_y$, wówczas musimy sprawdzić na którym polu znajduje się segment $<S_1..S_j>$ np. na Polu x, i następnie wykonać operację:

$$<S_1..S_j> \rightarrow Pole_y = <S_1..S_{j-1}> \rightarrow Pole_z,\ S_j \rightarrow Pole_y,\ <S_1..S_{j-1}> \rightarrow Pole_y,$$

gdzie $z \neq y$ i $z \neq x$.

Poniżej przedstawimy w pseudo-kodzie algorytm tworzący wieże Hanoi:

```
Move(i, Pole_x, Pole_y)
{
        Pole_z = {Pole_x, Pole_y} ∩ {1, 2, 3}
        if i=1 then
                S1 → Pole_y

        if i=2 then
                S1 → Pole_z
                S2 → Pole_y
                S1 → Pole_y

        else
                Move(i-1, Pole_x, Pole_z)
                Si → Pole_y
                Move(i-1, Pole_z, Pole_y)
}
```

Dla N segmentowej wieży, rozwiązaniem jest ciąg operacji elementarnych wyznaczonych przez wywołanie procedury:

Move(N,1,3)

Jako że poszczególne wywołania funkcji Move zajmują ten sam czas obliczeń „k", złożoność obliczeniowa całego algorytmu zależy od wielkości N. W tym przypadku ilość wywołań funkcji Move, to 2^{N-1}, więc złożoność algorytmu, to $O(k*2^{N-1})$.

9.4 Algorytmy zachłanne

Metoda zachłanna znajduje szczególne zastosowanie wszędzie tam, gdzie zachodzi konieczność szybkiego znalezienia nie koniecznie najlepszego rozwiązania. O jakości rozwiązań znajdowanych tą metodą powiedzieć możemy tylko tyle, że nie są one najgorsze, a w miarę zbliżania się do rozwiązań optymalnych zasadniczo wydłuża się czas obliczeń. Może się zdarzyć, że algorytm zachłanny znajdzie rozwiązanie, które będzie najlepsze, jednak będzie to raczej szczęśliwy traf, niż zamierzenie projektanta algorytmu.

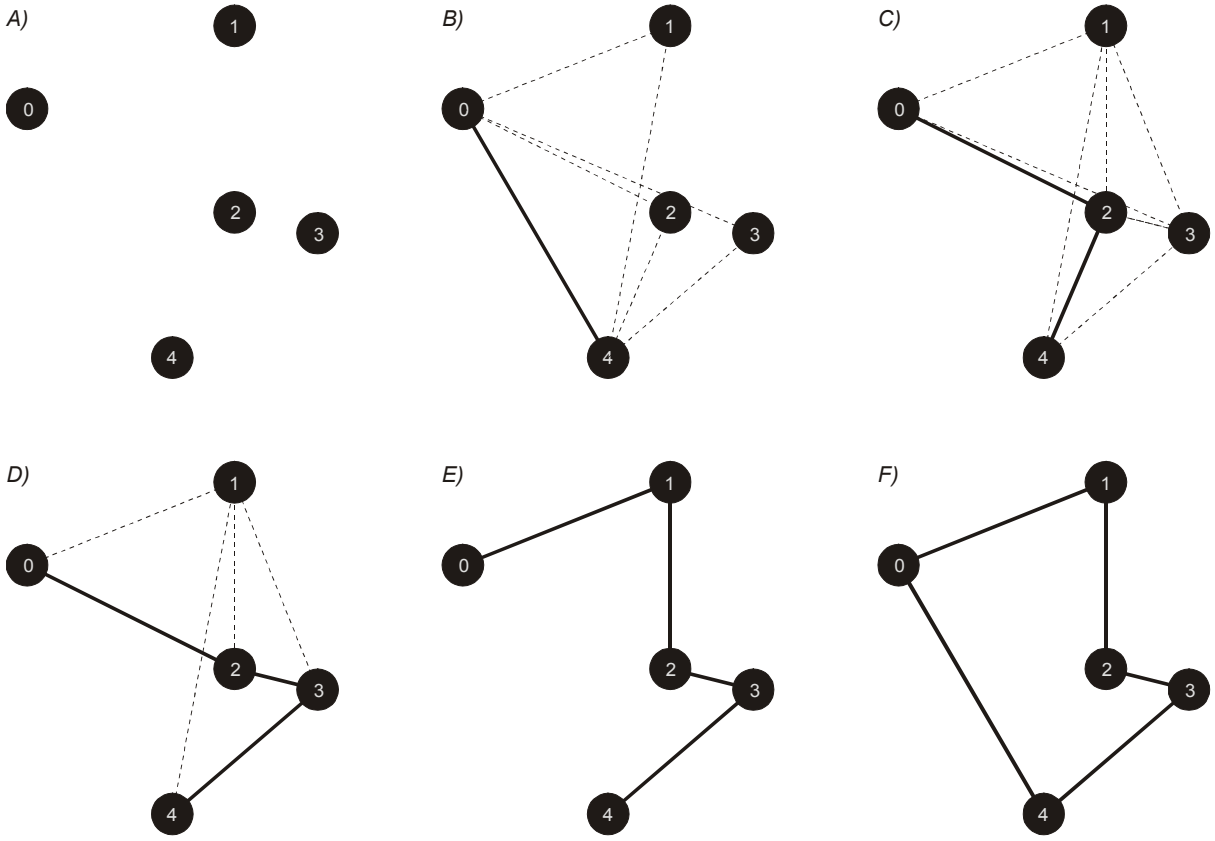
Niektóre algorytmy zachłanne po znalezieniu pierwszego rozwiązania stosują optymalizację rozwiązania, by zbliżyć je do najlepszego.

W celu przeanalizowania działania algorytmu zachłannego rozpatrzymy jeszcze raz problem komiwojażera. Tym razem będzie to komiwojażer

poruszający się po terenie „otwartym”, dlatego do pomiaru odległości użyjemy metryki euklidesowej. Jego zadanie będzie polegało na odwiedzeniu N miejsc poczynając od dowolnego miejsca. Podobnie jak wcześniej, będzie on mógł daną krawędzią grafu przejść tylko raz. Mając do czynienia z N miejscami, możemy podać ich współrzędne:

$M_1 = (x_1, y_1)$
$M_2 = (x_2, y_2)$
$M_3 = (x_3, y_3)$
...
$M_N = (x_3, y_3)$
Koszt przejścia krawędzią $(M_i, M_j) = d(i,j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$

Algorytm wybiera najpierw dowolną krawędź łączącą dwa miejsca. Jest to początkowa marszruta komiwojażera. W następnym kroku stara się włączyć do tej marszruty kolejny wierzchołek, by całkowita jej długość wzrosła jak najmniej (metryka euklidesowa spełnia warunek trójkąta, więc długość marszruty nigdy nie zmaleje przy włączaniu do niej nowego wierzchołka). Algorytm kończy działanie, gdy wszystkie wierzchołki są włączone do marszruty. Na rysunku nr 24 pokazany jest proces generowania marszruty dla 5 wierzchołkowego grafu:



Rysunek 24 Zachłanne generowanie marszruty komiwojażera

Patrząc na proces poszukiwania marszruty od razu widać, iż jest on zachłanny, gdyż algorytm stara się w danym momencie dołączyć do ścieżki nieodwiedzony dotąd wierzchołek w sposób zachłanny – przez wybór najkrótszych krawędzi do niego prowadzących. Taka strategia może, ale nie musi doprowadzić do optymalnego, końcowego rozwiązania.

Przebieg procesu generowania marszruty przedstawia poniższy algorytm napisany w pseudo języku:

Legenda:
Visited – Zbiór odwiedzonych wierzchołków
NonVisited – Zbiór nieodwiedzonych wierzchołków
Path – Zbiór krawędzi należących do marszruty komiwojażera.
AttachCost – Minimalny koszt dołączenia do aktualnej marszruty kolejnego wierzchołka
MinVertex – Wierzchołek, który zostanie dołączony do marszruty
MinEdge – Krawędź, która zostanie usunięta z marszruty
MinEdge|1
MinEdge|2 Wierzchołki krawędzi, która zostanie usunięta z marszruty
NonVisited = $\{1...N\} \setminus \{v_1, v_2\}$ v_1, v_2 = dowolnie wybrane wierzchołki
Path = $\{(v_1, v_2)\}$
While NonVisited $\neq \emptyset$ do
    AttachCost = Max
    //pętla po krawędziach marszruty
    For each edge $(v_i, v_j) \in$ Path do
        //pętla po nieodwiedzonych wierzchołkach
        For each vertex $v_k \in$ NonVisited do
            If AttachCost > $-d(v_i, v_j) + d(v_i, v_k) + d(v_j, v_k)$ then
                AttachCost = $-d(v_i, v_j) + d(v_i, v_k) + d(v_j, v_k)$
                MinVertex = v_k
                MinEdge = (v_i, v_j)

    Path := (Path \ {MinEdge}) $\cup$
        {(MinEdge|1, MinVertex), (MinVertex, MinEdge|2)}
    NonVisited := NonVisited \ {MinVertex}

Algorytm działa krokowo dopóki są jeszcze nieodwiedzone wierzchołki. W każdym kroku analizuje każdą krawędź marszruty pod kątem możliwości zastąpienia ją dwoma innymi krawędziami przechodzącymi przez nie odwiedzony dotychczas wierzchołek. Znalezienie najlepszej marszruty jest problemem NP- zupełnym, natomiast złożoność przedstawionego algorytmu jest rzędu $O(n^2 * |E|)$, gdzie |E| jest ilością krawędzi w grafie komiwojażera.

Rozwiązanie uzyskane metodą zachłanną można poddać dalszej lokalnej optymalizacji. Możemy np. podzielić dzielić marszrutę na mniejsze segmenty, i

badać, czy zamiana kolejności odwiedzania wierzchołków wewnątrz wybranego segmentu wpłynie na polepszenie rezultatu.

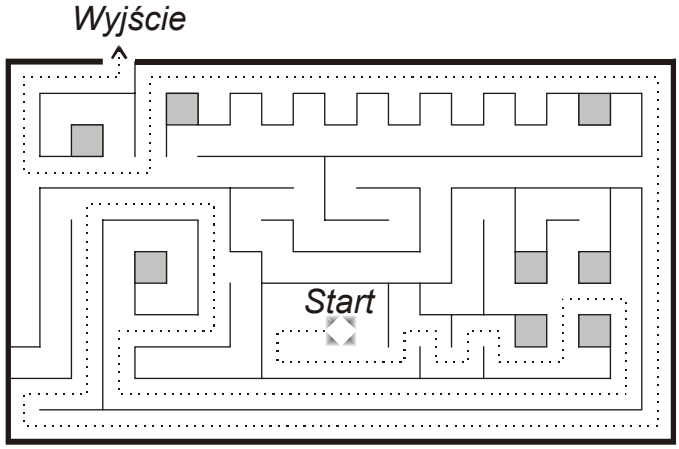
9.5 Algorytmy z nawrotami

Algorytmy z nawrotami używane są do znajdowania najlepszego rozwiązania danego problemu. Polegają one na poszukiwaniu rozwiązania przez stopniowe jego budowanie w kolejnych stanach algorytmu lub redukowanie, gdy nie prowadzi ono do celu.

Będąc w konkretnym stanie, algorytm z nawrotami stara się zbliżyć do rozwiązania na wszelkie możliwe sposoby. Jeśli możliwe jest rozbudowanie rozwiązania w aktualnym stanie i jeżeli to rozbudowanie nie było jeszcze analizowane, wówczas algorytm przechodzi do nowego stanu, zawierającego omawiane rozbudowanie. Jeśli w aktualnym stanie żadne rozbudowanie nie jest wykonalne, lub wszystkie możliwe rozbudowania zostały przeanalizowane, wtedy algorytm cofa się do poprzedniego stanu.

Tworzenie rozwiązania przypomina przemieszczanie się po drzewie począwszy od korzenia. W miarę rozbudowywania rozwiązania posuwamy się w stronę liści, które mogą być ostatecznym rozwiązaniem lub „ślepą uliczką". Jeśli w aktualnym węźle wszystkie dzieci zostały przeanalizowane, wtedy cofamy się do rodzica tego węzła i odwiedzamy pozostałe dzieci tego rodzica itp.

W celu zobrazowania metody z nawrotami przeanalizujemy proces szukania wyjścia z labiryntu. Labirynt będzie umieszczony w prostokącie M x N. Pomiędzy każdymi dwiema sąsiadującymi lokalizacjami może być przejście, lub może być zagradzająca je ściana. Rysunek nr 25 przedstawia przykładowy labirynt możliwy do przejścia:



Rysunek 25 Labirynt i metoda z nawrotami

Załóżmy, że posiadamy już odpowiednią strukturę danych reprezentującą taki labirynt, oraz że ma ona funkcję PATH(x1, y1, x2, y2), która zwraca prawdę, jeśli istnieje w labiryncie przejście pomiędzy dwiema sąsiadującymi lokalizacjami (x1, y1) → (x2, y2), lub fałsz, jeśli pomiędzy tymi lokalizacjami jest ściana.

Algorytm z nawrotami, który będzie realizował szukanie wyjścia z labiryntu będzie wykorzystywał stos S i tablicę odwiedzonych lokalizacji rozmiaru M x N oznaczoną jako VISITED. Stos S będzie zawierał kolejne lokalizacje reprezentujące ścieżkę od lokalizacji startowej do lokalizacji aktualnej. Tablica VISITED będzie gwarantem, że algorytm nie przejdzie do wcześniej odwiedzonej lokalizacji.

W każdym kroku działania algorytmu, stara się on przejść do nie-odwiedzonej dotąd lokalizacji. W faktu, iż agent przybywa do tej lokalizacji z innej (już odwiedzonej) wynika, że ma do wyboru jeden z maksymalnie trzech kierunków. Jeśli funkcja PATH pozwala na pójście w jednym z kierunków i przyszła lokalizacja FL nie była jeszcze odwiedzona, wówczas aktualna lokalizacja zapamiętywana jest w tablicy VISITED i odkładana na stos S, po czym aktualną lokalizacją staje się FL i algorytm przechodzi do następnego kroku.

Jeśli nie możliwe jest przejście do nowej lokalizacji, wówczas ze stosu pobierana jest wartość, która staje się aktualną lokalizacją. Jeśli przejście do nowej lokalizacji nie jest możliwe, a stos jest pusty, to rozwiązanie nie istnieje.

Poniżej przedstawiony jest kod algorytmu z nawrotami zapisany w pseudo-języku:

(Użyte zostały funkcje operujące na stosach: PUSH(S,x) – umieszczenie elementu x na stosie S, oraz x = POP(S) – przypisanie x wartości ze szczytu stosu S, i usunięcie tego elementu ze szczytu stosu).

```
Start:
Boolean Move = true
Boolean[M][N]  VISITED
Stack S = ∅
For i=1 to M do
        For j=1 to N do VISITED[i][j] = false
StartLocation = (x_o, y_o)
FinishLocation = (x_f, y_f)
ActLoc = StartLocation          //ActLoc  oznacza aktualną lokalizację
While(Move) {
        VISITED[ActLoc_x][ActLoc_y] = true
        If ActlLoc = FinishLocation then
                PUSH(S, ActLoc)
                 return S
                                                        //przejście ←
```

```
        Else If (ActLocx>1) and VISITED[ActLocx-1][ActLocy] = false
        and PATH(ActLocx,ActLocy, ActLocx-1,ActLocy) then
                PUSH(S,ActLocx,ActLocy)
                ActLocx = ActLocx-1
                                                        //przejście →
        Else If (ActLocx<M) and VISITED[ActLocx+1][ActLocy] = false
        and PATH(ActLocx,ActLocy, ActLocx+1,ActLocy) then
                PUSH(S,ActLocx,ActLocy)
                ActLocx = ActLocx+1
                                                        //przejście ↑
        Else If (ActLocy>1) and VISITED[ActLocx][ActLocy-1] = false
        and PATH(ActLocx,ActLocy, ActLocx,ActLocy-1) then
                PUSH(S,ActLocx,ActLocy)
                ActLocy = ActLocy-1
                                                        //przejście ↓
        Else If (ActLocy<N) and VISITED[ActLocx][ActLocy+1] = false
        and PATH(ActLocx,ActLocy, ActLocx,ActLocy+1) then
                PUSH(S,ActLocx,ActLocy)
                ActLocy = ActLocy+1

        Else If S≠ ∅ then
                ActLoc = POP(S)

        Else Move = false
}
```

10. Pytania kontrolne

1. Posługując się zasadą indukcji udowodnij, że każde drzewo binarne posiadające n liści składa się z 2n-1 węzłów
2. Udowodnij równoważność definicji: gramatycznej równowagi nawiasów oraz równowagi nawiasów poprzez skanowanie.
3. Zbuduj automat skończony akceptujący liczby rzeczywiste w zapisie zmienno-przecinkowym.
4. Zdefiniuj automat skończony i wyjaśnij definicję na dowolnym przykładzie
5. Zbuduj automat skończony oparty na alfabecie binarnym akceptujący słowa o nieparzystej liczbie 0 i 1.
6. Zdefiniuj rozszerzenie tranzytywne funkcji przejścia automatu skończonego
7. Skonstruuj niedeterministyczny automat skończony oparty na alfabecie {1,2,3}, który akceptuje tylko takie słowa, w których pomiędzy ostatnim symbolem słowa a wcześniejszym wystąpieniem tego *symbolu* są tylko symbole większe, np.: ·····12322321, ·····23332, ·····33.
8. Podaj formalną definicję niedeterministycznego automatu skończonego oraz języka, który jest przez niego akceptowany
9. Udowodnij równoważność deterministycznego i niedeterministycznego automatu skończonego.
10. Skonstruuj deterministyczny automat skończony oparty na alfabecie {1,2,3}, który akceptuje tylko takie słowa, w których pomiędzy ostatnim symbolem słowa a wcześniejszym wystąpieniem tego *symbolu* są tylko symbole mniejsze, np.: ·····11, ·····2112, ·····32211213.
11. Zademonstruj na dowolnym przykładzie działanie algorytmu ε-eliminacji dla automatów skończonych.
12. Zademonstruj na dowolnym przykładzie działanie algorytmu minimalizacji dla automatów skończonych.
13. Podaj formalną definicję wyrażenia regularnego i zastosuj ją dla dowolnego przykładu.
14. Przedstaw ogólny algorytm konstrukcji wyrażenia regularnego na podstawie automatu skończonego.
15. Przedstaw ogólny algorytm konstrukcji automatu skończonego na podstawie wyrażenia regularnego.
16. Skonstruuj automat skończony dla wyrażenia regularnego $(01(0+1))^*$ korzystając z algorytmu konstrukcji automatu skończonego na podstawie wyrażenia regularnego.
17. Sprawdź równoważność wyrażeń regularnych $(0+1)^*$ oraz $(0^*1^*)^*$.
18. Udowodnij lemat o pompowaniu dla języków regularnych.
19. Stosując lemat o pompowaniu sprawdź, czy język $\{a^n b^n \mid n \in \aleph\}$ jest regularny.

20. Sprawdź, czy jeżeli L_1 oraz L_2 są regularne, to L_1L_2, $L_1 \cup L_2$ oraz L_1^* są regularne.
21. Udowodnij, że jeśli L jest regularny, to L^R też jest regularny.
22. Podaj jak sprawdzić problem przynależności, pustości oraz skończoności języka regularnego.
23. Podaj formalną definicję gramatyk bez-kontekstowych i zademonstruj ją na dowolnym przykładzie.
24. Podaj lewo i prawostronny wywód słowa 0110011 w gramatyce ({S,A}, {0,1}, {S→ SA | ε, A → 0A1 | A1 | 01} ,S).
25. Udowodnij twierdzenie o równoważności drzew wywodów i wywodów lewostronnych / prawostronnych.
26. Podaj formalną definicję automatu ze stosem i zademonstruj ją na wybranym przykładzie.
27. Zbuduj automat ze stosem akceptujący język $\{a^nb^n \mid n \in \aleph\}$
28. Zbuduj automat ze stosem akceptujący język generowany przez gramatykę ({S,A}, {0,1}, {S→ SA | ε, A → 0A1 | A1 | 01} ,S).
29. Udowodnij, że jeśli istnieje automat ze stosem A_1 rozpoznający język L przez pusty stos, to da się dla niego skonstruować automat A_2 rozpoznający język L przez stan końcowy.
30. Udowodnij, że jeśli istnieje automat ze stosem A_1 rozpoznający język L przez stan końcowy, to da się dla niego skonstruować automat A_2 rozpoznający język L przez pusty stos.
31. Udowodnij, że dla dowolnej gramatyki bez-kontekstowej G da się stworzyć automat ze stosem A taki, że L(A) = L(G).
32. Udowodnij, że dla dowolnego automatu ze stosem A da się stworzyć gramatykę bez-kontekstową G taką, że L(G) = L(A).
33. Podaj charakterystyczne cechy deterministycznych automatów ze stosem.
34. Omów algorytm usuwania symboli bezużytecznych z gramatyk bez-kontekstowych na przykładzie gramatyki G=({S,A,B,C,D,E}, {0,1}, {S→AB|C|DED; A→0B|C; B→1|A0; C→AC|C1; E→0}, S).
35. Omów na dowolnym przykładzie algorytm pozbycia się ε-produkcji dla gramatyk bez-kontekstowych.
36. Omów na dowolnym przykładzie algorytm pozbycia się produkcji zbędnych dla gramatyk bez-kontekstowych.
37. Podaj definicję postaci normalnej Chomsky'ego dla gramatyk bez-kontekstowych oraz pokaż jak dowolną gramatykę przekształcić do tej postaci.
38. Sprowadź do postaci normalnej Chomsky'ego gramatykę G=({S}, {0,1}, {S→0S1 | 01}, S).
39. Udowodnij lemat o pompowaniu dla gramatyk bez-kontekstowych.
40. Korzystając z lematu o pompowaniu dla gramatyk bez-kontekstowych sprawdź, czy język $L=\{a^nb^{2n}c^n \mid n \in \aleph\}$ jest bez-kontekstowy.

41. Udowodnij twierdzenie o podstawieniu dla gramatyk bez-kontekstowych.
42. Udowodnij, że rodzina języków bez-kontekstowych jest domknięta ze względu na: sumę, konkatenację, powielanie (operator *) oraz odwrócenie (operator $^{\leftarrow}$).
43. Sprawdź, czy rodzina języków bez-kontekstowych jest domknięta ze względu na przecięcie.
44. Sprawdź, czy rodzina języków bez-kontekstowych jest domknięta ze względu na dopełnienie.
45. Podaj jak sprawdzić, czy dany język bez-kontekstowy jest skończony
46. Przedstaw na wybranym przykładzie algorytm sprawdzający przynależność słowa do języka bez-kontekstowego.
47. Przedstaw podział gramatyk zaproponowany przez Chomsky'ego.
48. Udowodnij zawieranie się klas języków: $L_3 \subset L_2 \subset L_1 \subset L_0$.
49. Podaj formalną definicję maszyny Turinga i zademonstruj ją na wybranym przykładzie.
50. Zdefiniuj pojęcie stanu i funkcji przejścia maszyny Turinga; Zilustruj te pojęcia dowolnymi przykładami.
51. Udowodnij równoważność akceptacji przez stan końcowy oraz zatrzymanie dla maszyn Turinga.
52. Przedstaw jak realizować wymianę symboli na taśmie maszyny Turinga.
53. Przedstaw jak realizować symulację wielu ścieżek sklejonych równolegle na jednej taśmie maszyny Turinga.
54. Przedstaw jak realizować symulację wielu taśm na jednej taśmie maszyny Turinga.
55. Przedstaw jak na deterministycznej maszynie Turinga symulować niedeterministyczną maszynę Turinga.
56. Udowodnij, że równoważność maszyny wielo-stosowej (automatu z wieloma stosami) z maszyną Turinga.
57. Udowodnij, że równoważność 2-licznika z maszyną Turinga.
58. Podaj definicję schematu minimalizacji pozwalającego na tworzenie nowych funkcji częściowych oraz ilustrujący ją przykład.
59. Podaj definicję schematu rekurencji prymitywnej oraz 3 funkcji bazowych.
60. Posługując się schematem rekurencji prymitywnej stwórz funkcje reprezentujące: dodawanie, mnożenie, poprzednika liczby, zera oraz odejmowania symetrycznego.
61. Podaj sposoby konstrukcji funkcji rekurencyjnie prymitywnych.
62. Udowodnij, że zbiory skończone są rekurencyjnie prymitywne.
63. Udowodnij, że zbiór potęgowy $A=\{1, m, m^2, m^3, ... \mid m>1\}$ jest rekurencyjnie prymitywny.
64. Udowodnij, że istnieje odwzorowanie wzajemnie jednoznaczne (bijekcja) pomiędzy $\aleph$ oraz $\aleph \times \aleph$.
65. Zdefiniuj funkcję Ackermanna na wybranym przykładzie.

66. Udowodnij, że bazowe funkcje dla rekurencji prymitywnej mogą być symulowane na maszynie Turinga.
67. Udowodnij, że każda funkcja częściowa, obliczalna przez maszynę Turinga jest rekurencyjna.
68. Podaj definicje i aksjomaty rachunku λ.
69. Podaj ilustrację drzewiastą λ-wyrażenia.
70. Przedstaw definicję notacji Θ(n), Ω(n), O(n) oraz wyraź złożoność obliczeniową dowolnej funkcji w każdej z tych notacji.
71. Przedstaw definicje i wzajemne relacje między klasami złożoności: P, NP, NP-zupełna.
72. Podaj przykłady problemów NP-zupełnych i przedstaw metodę dowodzenia NP-zupełności problemów.
73. Przedstaw metodę Branch & Bound i zastosuj ją do rozwiązania dowolnego problemu.
74. Przedstaw metodę programowania dynamicznego i zastosuj ją do rozwiązania dowolnego problemu.
75. Przedstaw metodę dziel i zwyciężaj i zastosuj ją do rozwiązania dowolnego problemu.
76. Przedstaw metodę zachłanną i zastosuj ją do rozwiązania dowolnego problemu.
77. Przedstaw metodę z nawrotami i zastosuj ją do rozwiązania dowolnego problemu.

11. Zakończenie

W pracy zostały przedstawione podstawowe elementy informatyki teoretycznej, takie jak języki formalne, automaty, modele obliczeń oraz klasy algorytmów. Zarówno algorytmy jak i modele obliczeń oparte są na konkretnych językach, stąd część pracy jest poświęcona teorii języków i automatów. Analizując początkowo języki proste, dochodzimy ostatecznie do języków, na które nie są nałożone żadne ograniczenia. Dla takich języków, istnieje model obliczeniowy je rozpoznający – maszyna Turinga. Języki rozpoznawane przez maszynę Turinga z powodzeniem mogą definiować prymitywne funkcje informatyczne, które są prekursorami zaawansowanych programów komputerowych.

Algorytmy najczęściej budowane są z segmentów, które należą do jednej z klas przedstawionych w pracy. Skoro każdy algorytm lub model obliczeń zapisany jest w konkretnym języku, maszyna Turinga teoretycznie może go symulować. W konsekwencji nawet najbardziej skomplikowane programy jak systemy operacyjne mogłyby być realizowane przy pomocy taśmy z danymi i prostej głowicy, która dokonując odczytu / zapisu w komórkach taśmy realizuje ściśle określony przez swoją funkcję przejścia schemat działania.

Praca skupia się głównie na teorii, dlatego możliwe jest jej rozbudowywanie przez dodawanie nowych przykładów bądź budowanie systemów informatycznych realizujących przedstawione modele obliczeń. Zamieszczone w pracy algorytmy nadają się do implementacji na prawdziwym komputerze. Przedstawiono więc szereg zadań programistycznych, które w połączeniu z pytaniami kontrolnymi mogą stanowić istotną wartość dydaktyczną dla studentów informatyki.

Bibliografia

1. BIGGS N., “Discrete Mathematics” Clarendon Pr, Revised edition 1993
2. BRASSARD G., BRATLEY P., “Algorithmics: Theory and Practice” Prentice-Hall, 1988
3. CAVANAGH J. J. F., “Digital Computer Arithmetic”. McGraw-Hill, 1984.
4. CORMEN T.H. , LEISERSON C. E., RIVEST R. L., „Introduction to Algorithms“, Massachusetts Institute of Technology, 1990
5. FORYŚ M., FORYŚ W., Teoria automatów i języków formalnych - książka w wersji elektronicznej przygotowywana aktualnie do druku
6. GRUSKA J., Foundations of computing, Thompson, 1997
7. HOPCROFT J. E., ULMAN J. D., Introduction to automata theory, languages and computing, Addison-Wesley, 1979
8. HOWIE J. M., Automata and Languages, Clarendon Pr; 1996
9. MARECKI J., „Algorytmy Numeracji i Grafów” Seminarium na temat „Badania Operacyjne” w ramach Beskidzkiego Festiwalu Nauki 2001
10. MARECKI J., „Metody Sortowania” Seminarium na temat „Podstawy informatyki i sieci komputerowych” w ramach Beskidzkiego Festiwalu Nauki 2001
11. MARECKI J., „Metody Sztucznej Inteligencji” Wydawnictwo Pracowni Komputerowej Jacka Skalmierskiego, Gliwice 2000
12. MARECKI J., „Modele Grafów” Seminarium na temat „Badania Operacyjne” w ramach Beskidzkiego Festiwalu Nauki 2001
13. MARECKI J., „Struktury Danych” Wydawnictwo Pracowni Komputerowej Jacka Skalmierskiego, Gliwice 2000
14. MARECKI J., „Teoria Funkcji Informatycznych” Seminarium na temat „Podstawy informatyki i sieci komputerowych” w ramach Beskidzkiego Festiwalu Nauki 2001
15. MARECKI J., „Teoria Funkcji Informatycznych” Seminarium na temat „Podstawy informatyki i sieci komputerowych” w ramach Beskidzkiego Festiwalu Nauki 2001
16. MARECKI J.,, Grafy i Rekurencje, Wydawnictwo pracowni komputerowej Jacka Skalmierskiego, 2002
17. OHO A.V., HOPCROFT J.E., ULLMAN J.D., “Projektowanie i analiza algorytmów komputerowych” PWN, Warszawa 1983
18. PIN J.E., Varieties of Formal Languages, Plenum Press, 1986
19. SALOMAA A., Computation and Automata, Cambridge Univ.Press, 1985
20. WIRTH N., “ Algorithms + Data Structures = Programs ”, Prentice-Hall, 1976

21. XUONG N. H., “Mathematiques Discretes et Informatique”, Masson, Paris 1991